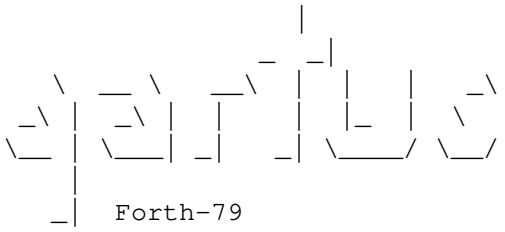
qartus Manual

Antonio Maschio
with the help of Ian Jones and Bruce Axtens

ABSTRACT



Welcome to the world of qartus! qartus is a Forth-79 interpreter with Double-Number Standard Extensions, based on the Forth-79 Standard (1980).

This manual is not a Forth primer; I assume you have the most common knowledge about programming in Forth; this is more like a reference manual that helps in using correctly the various dictionary words and understand the memory state. You could find also useful the qartus man page for the console execution options, other more general info and the installing instructions.

I here thank Ian and Bruce, the ones who shared with me this creation. They discovered many bugs, suggested many things, cleared many concepts, tested everything, showing great patience throughout the whole process and supporting my discouragements. They helped me to interpret some features of the Forth-79 Standard. They are friends, though we've never met: all of our work has been shared through emails, but our friendship is anyway strong. Thank you, Ian and Bruce.

I don't belong to any Forth circle, no Forth group, no Forth organization; no university, no school, no structure. And I'm not a professional programmer, so please don't tell me "you should've done this that way and that this way!" because I know that my programming style is pretty naive. I just want my algorithms to work: Speeding them up and making them beautiful are tasks that go beyond my interests.

And since I make errors (as anybody), readers are encouraged to report all errors and inconsistencies (both in the program and in the manual) to:

ing dot antonio dot maschio at gmail dot com

Sustain my efforts! Donations are welcome through Paypal to

tbin at libero dot it

Thanks.

26 December 2021

Per correr miglior acque alza le vele
omai la navicella del mio ingegno...
(Dante, Purgatorio, Canto I, 1472)

To run o'er better water hoists her sails
the little vessel of my genius now...
(Dante, Purgatory, Chant I,
transl. Courtney Langdon, 1920)

1. General notes

1.1. Introducing qartus

In late 1983 (or maybe early 1984) a good friend of mine, Alessandro (a.k.a. Sguinz)[1], once told me in the classroom, while we were waiting for the professor to come: "*I've just read about a new fantastic programming language!*". He grabbed a piece of paper and wrote something as

```
2 3 + DUP 4 SWAP
```

on it, explaining me the stack, the basic stack manipulation words, and in general how Forth works. This got me hooked. I began insisting with my mother to buy me a Forth book (any) and finally she agreed. The book of Alan Winfield was mine. I read it dozens of times, did all the exercises, tried to get into the complexity of the defining words, but what I lacked was a real environment. No computer, only theory.

Many years passed, and one day in 2011 I began to design my personal Forth compiler. The development soon stopped, because of my job and also because I had begun a carrier in resurrecting old forgotten programming languages by writing interpreters for them (BASIC and ALGOL interpreters, mainly). In 2017 I resurfaced the project, developed it a bit more and finally, the Alpha version of qartus is out.

Why Forth? Maybe because I like programming, and I like Forth and I like the idea to create something by building small pieces of it. But principally because Forth is the most fascinating imperative language ever created, in my opinion.

Why Forth-79? Because it's the First Standardization of the Forth language. Because it's simple. Because the first Forth version I studied was Forth-79.

qartus is a **gcc** program for Linux or UNIX (compilable also by `clang`), it's completely free and it's distributed according to the GNU Public License version 3. You are welcome if you want to use it, and you are welcome even if you don't want. I only ask you to tell me if you find errors and inconsistencies: I will try to solve them as soon as possible.

The name qartus comes from the Latin word Quartus, meaning 'fourth'. And since Moore removed the letter 'u' from 'fourth', giving birth to forth, could I do less?

Final note: I wrote this book using vim/gvim and groff. If you find mistakes and bad formatting, please write to me. I'd like to correct all of them.

I'm the only responsible for the things written in this manual. The directory `docs-src/` of the package contains all the files needed to recompile the manual in case you lose it, provided you have `groff` and `ps2pdf` installed on your system. Simply enter this directory and type `make manual`:

```
/path/to/qartus-src/ $ cd docs-src
/path/to/qartus-src/docs-src $ make manual
```

and a fresh new copy of the pdf manual will be available to you. You can also download a free copy from the home page of qartus. You are free to choose the option you want.

[1] I know this nickname is hard to pronounce for people out of Veneto, but it sounds like the English word *'swince'* with a `'g'` dur between `'s'` and `'w'` (the `'g'` of `'get'`).
More here: `https://www.urbandictionary.com/define.php?term=Swince`.

1.2. The documents

qartus is an implementation of Forth-79, following the October 1980 document by the Forth Standard Team. Its aim is to give to the qartus user the least but most complete number of words for programming, with the ambiguities of the 1979 Standard resolved. I don't know if I succeeded, really, in this huge task.

- The Forth-79 manual can be downloaded here:

  `https://www.complang.tuwien.ac.at/forth/standards/Forth-79.pdf`

  This document is included in the qartus package.

  A textual document with the same text can be found here:

  `https://www.complang.tuwien.ac.at/forth/fth79std/FORTH-79.TXT`

  qartus's reference manual is modeled on this document.

- To create qartus I have read again (for the 100th time or so) the wonderful book of Alan Winfield "*The complete Forth*" (Italian edition, titled "*Forth - corso di programmazione per microcalcolatori*"), 1983, which belongs to my books collection since then, when I was a boy. The compiler used by Mr. Winfield was R-Forth, maybe a Forth-79 with some fig-Forth issues (for instance `SIGN`, `DPL`), or maybe a fig-Forth with some Forth-79 insertions... who knows...

- The fig-Forth manual too was a source of information. I've read the 1986 book "*Extended fig-Forth rev. 1*"; you can download it here:

  `http://www.atarimania.com/documents/Exteneded_Fig_Forth_rev_1.pdf`

  The fig-Forth manual (release 1) is available here:

  `http://www.cs.drexel.edu/~bls96/6809sbc/doc/figdoc/glossary.txt`

  The fig-Forth is the elder cousin of Forth-79.

- The two articles "*Upgrading Forth-79 Programs*" about how to translate code written for Forth-79 to Forth-83, available in "*Forth Dimensions*" Vol. VI, N. 3 September/October 1984 and dowloadable at

  `http://www.forth.org/fd/FD-V06N3.pdf`

  and "*Forth-83 Program to Run Forth-79 Code*" in "*Forth Dimensions*" Vol. VI, N. 4 November/December 1984, downloadable at

  `http://www.forth.org/fd/FD-V06N4.pdf`

  were also useful to understand the real nature of Forth-79.

- An important resource file was the manual of Hartforth, a Forth-79 engine for the TRS80 computer, available at

  `http://tim-mann.org/trs80/doc/forth.pdf`

- Of course I couldn't continue without mentioning the two imperative books by Leo Brodie, "*Starting Forth*" (1981), substantially written in fig-Forth in the book version (and ANSI-fied in the online version), and "*Thinking Forth*" (1984), written in Forth-83 (and ANSI-fied in the online version). The first is available online here:

  Original scanned version (fig-forth side)
  `https://1scyem2bunjw1ghzsf1cjwwn-wpengine.netdna-ssl.com/wp-content/uploads/2018/01/Starting-FORTH.pdf`

  ANSI-fied version (ANSI 1994 side)
  `https://www.forth.com/starting-forth/`

  while the second can be downloaded (only as the ANSI 1994 side, AFAIK) as a pdf file here:

  `https://www.forth.com/forth-books/`

  (scroll down to see the links); the pdf file is good if you have no latex system on your UNIX machine (or if you have no UNIX machine), otherwise you can download the latex files and compile.[2]

Enough? No! I consulted many other specialized texts, here listed:

- Of course, the first and only Forth Handbook is the Conklin/Rather "*Forth Programmer's Handbook*"; I have a printed copy of the First Edition, October 2001. Of course it refers to the ANSI 1994 Forth, but it's a wonderful and clear source of information. You can find a more recent version on Internet (Searching for "Forth Programmer's Handbook").

- Some ideas came from "*Forth on the BBC microcomputer*", by Richard De Grandis Harrison, January 17, 1983 (where I grabbed some ideas for the Screen Editor above all). This book was a great mine of information in all cases where the Standard rules were not clear. The book, published by Acornsoft, is available at

  `http://stardot.org.uk/mirrors/www.bbcdocs.com/filebase/software/apps/forth-on-the-bbc-microcomputer.pdf`

- Some other ideas came from the manual "*microForth Primer*", second Edition, by the Forth, Inc., dating back to August 1978 (with Elizabeth Rather behind the book, I suppose), and that can be found (along with other documentation about microForth) here:

  `https://www.forth.com/resources/archive-forth-systems-software/`

- The manual of the Max-Forth for the 68HC11 (a ROM-based operating system and language) was indeed very very useful; you can find it here:

  `http://www.science-info.net/pages/GordonCouger/stuff/68hc11/1FORTH-programs/MASTERV3.ASC`

  A glossary of Max-Forth is also available here:

  `http://www.newmicros.com/FORTH/HC11/V35/Alphabetical/Detailed/index.html`

[2] I'm so lucky to have an original printed copy of it (Forth-83 side), which is invaluable, for me. A sort of secular bible I turn to from time to time.

- The manual of the tForth (Jun 1987), by Jef Raskin, is a wonderful conceived textual document, that can be found here[3]:

  `http://www.canoncat.net/cat/Cat%20tForth%20Documentation.txt`

- How not to mention the manual of PolyForth, by Elizabeth D. Rather, Leo Brodie and the Technical Staff of FORTH, Inc.? PolyForth was someway more than a Forth engine, and more like a Forth shell, the grandpa of SwiftForth. This manual gave also the base for the Forth Handbook (and later the sf manual). The pdf of the Fifth Edition is available here:

  `http://www.greenarraychips.com/home/documents/greg/DB005-120825-PF-REF.pdf`

- Kim Harris's Forth Course, dating back to June 1980 and using fig-Forth, was a good lecture (it's not a book, rather a collection of slides), and can be found here:

  `http://www.forth.org/KimHarris/KimHarris.html`

- The manual of the jforth, whose glossary is a huge effort (something I'll never be able to repeat...), can be found here:

  `http://www.jforth.org/TableOfContents.html`

- Have you ever read "*Forth Dimensions*"? It was a wonderful review, dedicated to Forth only. It's a pity for me (and maybe for some of you) that I became aware of it only after it ceased publications. In it, I found many programs (some of them were transcribed by me and included in qartus's package as block or files). The review can be found in many places; I use the site www.forth.org. Another source can be found at

  `https://www.complang.tuwien.ac.at/forth/forth-dimensions/`

- I cannot omit "*Forth for the complete Idiot*", by C.H. Ting, issued in 1983 and revised in 1984 (79-oriented, and targeted for the APPLE Forth). Here's an excerpt of the introduction: "*FORTH is a language of mystic quality. The learning process for many FORTH programmers can be described only in religious terms. However, what I consider to be the greatest myth about FORTH is that FORTH IS AN ELITIST'S LANGUAGE. This myth is perpetuated because most people think that FORTH is difficult to learn, FORTH programs are difficult to read and to comprehend, and that a FORTH computer is expensive, at least comparing to most other personal computers running BASIC. My opinion is that FORTH IS THE IDIOT'S LANGUAGE which is best suited for common people who are being shuffled into computer literacy, kicking and screaming. Why? Because FORTH is actually a very simple language to learn. It has the simplest grammar and the fewest syntax rules than any other computer languages. Above all, FORTH can be extended and modified so that you can use your own language or terms to converse with it. I was asked what is the qualification of a person to learn FORTH. My answer was that he must have at least one finger, to hit keys on a keyboard. Well, what else does he need to learn FORTH?*". You can find this essay here:

  `http://www.forth.org/Ting/Forth-for-the-Complete-Idiot/Forth-for-the-Complete-Idiot.pdf`

  Enjoy this minimalistic manual!

[3] I must confess that the original name of my project was tforth, but I didn't know it was used something like 30 years before. So, when I finally stumbled into this site, I had to change the project's name to qartus, not without a tip of sorrow. 't' stood for Tony, of course! But this is not the entire story! Read on.

- I must also cite this forth interpreter, called yforth, made by an Italian like me by the name of Luca Padovani in 1997 and developped until 2012. The name of yforth was the name that I'd been using for years, and it was only by chance that I discovered it was already in use[4]. You can find yforth at the following site:

  `https://packages.debian.org/sid/interpreters/yforth`

- Last (but not least), the marvellous guide to implement Forth-79 on a fig-Forth system, titled "*Forth-79 Standard Conversion*", by Robert L. Smith, dating back to 1981, and downloadable from Stackosaurus here:

  `http://www.stackosaurus.com/misc/Forth-79_Standard_Conversion.pdf`

  Behind the text lines you can see a huge brain work... The Stackosaurus site has also a page devoted to fig-Forth here:

  `http://www.stackosaurus.com/figforth.html`

I have also consulted reference manuals of the further evolutions of Forth, specially Forth-83, whose manual is available here:

`http://forth.sourceforge.net/std/fst83/`

Forth-83 was very useful to detect the real nature of a Forth-79 compiler, by comparing their different features.

Many others are available over the Internet, I couldn't survey them all.

Anyway, for any other technical Forth-related data you need to know, consult the site

`https://www.forth.com/resources/forth-programming-language/`

and read the Leo Brodie books! Better than anything else!

[4] Again! after tforth, after yforth, will maybe qartus be the last?

1.3. Definition of terms

These definitions clarify the qartus's environment terminology used into this manual.

`address, byte`

An unsigned number that locates the first of two 8-bit bytes in the memory address space over the range {0..65,535}. It is a virtual representation on a memory mapping over two bytes. Address arithmetic is modulo 65536 without overflow.

`address, code field`

The address of the first byte of memory in a word definition, where execution codes (in a colon-definition) are stored. This is the address returned by `FIND`.

`address, compilation`

The numerical value equivalent to a word definition, which is compiled for that definition. The address interpreter uses this value to locate the machine code corresponding to each definition.[5]

`address, parameter field`

The address of the first byte of memory associated with a word definition for the storage of compilation addresses (in a colon-definition), numeric data and text characters. This is the address returned by ' (tick).

`block`

The unit of data from mass storage, referenced by a block number. A block contains 1024 bytes. The translation from block number to device is done internally through UNIX calls, by mapping the storage block file content to a dedicated buffer in memory (see 'screen').
Block numbers range is {1..65535}. Block 0 is not available.

`block buffer`

A memory area where a mass storage block is maintained (see also 'screen').

`buffer`

See 'screen' and 'block buffer'.

`byte`

An assembly of 8 bits. With reference to memory, it is the storage capacity for 8 bits.

`cell`

A 16-bit memory location. The n-th cell contains the 2n-th and (2n+1)-th byte of qartus's address space. When represented on the stack, the number is retrieved as a whole 16-bit number. When in memory, the lower 8-bits are at the lower address.[6] For the definition of 'cell pair', see 'string'.

[5] The Standard specifies here: "*[It] may also be called the code field address*", but qartus uses a different method, compiling the Link Field Address (as a 32-bit number). The code field address can be derived from a fixed size distance from LFA (4 bytes + the name length). In any case, this is transparent to the user.

[6] I must note that in the Standard the byte order is expressively ignored (page 3: "*The byte order is presently unspecified*"). When I designed this feature for qartus, I decided to use the same pattern generally used in previous Forth-79 compilers, probably designed to ease the usage of C@ and C! directly on the address number. I must thank Ian Jones for pointing me out the problem, that I had initially addressed in the opposite way.

character

An 8-bit number which represents a terminal character. The ASCII character set is used. When contained in a larger field, the higher order bits are zero. 7 bits are taken into account for the default range {1..127}.

The correspondence with keyboard keys and ASCII control codes (in the range 1-31) are contained in the following table:[7]

Hex	ASCII	CTRL-Key	Hex	ASCII	CTRL-Key
0	NUL	(*)	10	DLE	CTRL-P
1	SCH	CTRL-A	11	DC1	(*)
2	STX	CTRL-B	12	DC2	CTRL-R
3	ETX	(*)	13	DC3	(*)
4	EOT	CTRL-D	14	DC4	CTRL-T
5	ENQ	CTRL-E	15	NAK	CTRL-U
6	ACK	CTRL-F	16	SYN	CTRL-V
7	BEL	CTRL-G	17	ETB	CTRL-W
8	BS	CTRL-H	18	CAN	CTRL-X
9	HT	CTRL-I	19	EM	CTRL-Y
A	LF	CTRL-J	1A	SUB	(*)
B	VT	CTRL-K	1B	ESC	(*)
C	FF	CTRL-L	1C	FS	(*)
D	CR	CTRL-M	1D	GS	CTRL-5
E	SO	CTRL-N	1E	RS	CTRL-6
F	SI	CTRL-O	1F	US	CTRL-7

(*) codes intercepted by the O.S. or by qartus and having a specific purpose.

The ASCII table for all normal printing characters in the range 32-127 are here reported:

02 gennaio 2021 Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
20		30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~

The range 128-255 depends on the Operating System and may vary from computer to computer and not represented here.

[7] Using KEY you can obtain these control ASCII codes, typing CTRL plus the relative key, except for those marked as (*).

Any character sequence in the input stream is case-sensitive only for the strings, which are sequences of characters enclosed in double quotes; any other sequence corresponding to Forth words is case-insensitive, included created-words names (using e.g. `VARIABLE`, `CREATE` etc..).

`compilation`

The action of accepting text words from the input stream and placing corresponding compilation addresses/codes in a new dictionary entry to be executed later.

`counted string`

See 'string'.

`defining word`

A word that, when executed, creates a new dictionary entry through compilation. The new word name is taken from the input stream. If the input stream is exhausted before the new name is available, an error condition exists. The built-in defining words are:

```
2CONSTANT 2VARIABLE : CONSTANT CREATE SET VARIABLE VOCABULARY
```

Any word that contains a defining word, or equivalently that contains another word defined via a defining word, is classified as a defining word.

`definition`

See 'word definition'.

`digit`

(definition by Robert L. Smith) A digit is any one of a set of ASCII characters which represent numeric values in the range from 0 to base-1. For bases greater than decimal 10, the set of characters is {0..9 A B C..Z} where the ascending ASCII sequence is used for A and above. The digit sequence is always case-insensitive.

`dictionary`

The set of word definitions in a computer memory. The dictionary entries are organized in vocabularies to enable location by name. The dictionary is extensible, growing toward high memory.

`division, symmetric`

Integer division in which the remainder carries the sign of the dividend or is zero, and the quotient is rounded toward zero. Note that, except for error conditions, the sequence `n1 n2 SWAP OVER /MOD ROT * +` is identical to `n1`.

Examples:

```
dividend divisor remainder quotient
   10       7        3         1
  -10       7       -3        -1
   10      -7        3        -1
  -10      -7       -3         1
```

error condition

An exceptional condition which requires action by qartus. The most general action is to display a message, returning control to the user.

The General Error Conditions (GCE) are:

1. input stream exhausted before a required <name>.
2. empty stack and full stack for the text interpreter for all explicit and implicit stacks.
3. an unknown word or an invalid number for the text interpreter.
4. compilation of incorrectly nested conditionals and loops.
5. interpretation of words restricted to compilation.
6. compilation of words restricted to interpretation.
7. FORGETting a dictionary word which is beyond the imposed FENCE.
8. insufficient remaining space in the dictionary or in the buffers.

In the built-in words Dictionary chapter, all situations where the error condition may manifest are reported; in all other unmentioned cases, the GCE previously defined takes place by emitting an error message and stopping the execution.

false

A zero number representing the false condition flag. All words returning a flag leave 0 as a 'false' flag.

flag

A number that has only two logical states, zero and non-zero. These are named 'true' = non-zero, and 'false' = zero. Words that test a condition leave 1 for true, 0 for false.

immediate word

A word defined to automatically execute when encountered during compilation, which handles exception cases to the usual compilation. See IF, DO, ." etc.

input stream

A sequence of characters available to the system, for processing by the text interpreter. The input stream is conventionally taken from a terminal (via the terminal input buffer where keyboard characters are stored) and mass storage (via a block buffer or via a file). >IN and BLK respectively specify the position in the input stream and the block number used to load the input stream. Words using or altering >IN and BLK are responsible for maintaining and restoring control of the input stream.

interpreter, text

The (set of internal) word definitions that repeatedly accept a word name from the input stream, locate the corresponding dictionary entry, and start the address interpreter to execute it. Text in the input stream interpreted as a number leaves the corresponding value on the data stack. When in compile mode, the addresses of Forth words are compiled into the dictionary for later interpretation by the address interpreter. Numbers are compiled, and when later interpreted place the data on the stack. Numbers are accepted as unsigned or negatively signed, according to BASE.

KiB unit

(from Wikipedia) The kibibyte is a multiple of the unit byte for quantities of digital information. The binary prefix *kibi* means 2^10, or 1024; therefore, 1 kibibyte is 1024 bytes. The unit symbol for the kibibyte is KiB.

load (block buffer)

The acceptance of text from a mass storage device and interpretation of the dictionary words, in the form of a block buffer. In current implementation, a block is by default 1024 characters wide (1 Kib), and it is loaded in one single pass and copied to the Input Buffer for interpretation when LOADed. This is the second general method for compilation of new definitions into the dictionary.

load (file)

The acceptance of text from a mass storage device and interpretation of the dictionary words, in the form of a textual file, specifying its name as the console argument of qartus's executable. In current implementation, each line of the file, which must not be greater than 1024 characters (1 Kib), is loaded and copied to the Input Buffer for interpretation, until the end of file. This is the first general method for compilation of new definitions into the dictionary.[8]

mass storage (block buffer)

Data is read from mass storage in the form of 1024 byte blocks in case of block buffers. The editing, saving, copying, altering and formatting of these blocks is accomplished either through the internal editors or outside of qartus. When indicated as UPDATEd (modified), data will be ultimately written to mass storage in 1024 file blocks located by default in $HOME/.qartus/blocks/, a directory created by the --config option (see also BLOCK). See the man page for the differences with the Windows© configuration.

mass storage (files)

Data is read from mass storage in the form of free-form textual files with preferred extension .ft. The editing, saving, copying, altering and formatting of these files is accomplished outside of qartus.

memory

The space where dictionaries, buffers and stacks are hosted, in a contiguous space in the range {0..65535}. The physical structure holding memory bytes is an array of 32-bits numbers, each called 'unit'. Each unit hosts the basic 8-bits byte capable of holding the higher or lower parts of a 16-bits number, the character in a buffer, the addresses and compilation codes (these last often in a full 32-bits format).

This structure is not perceivable by the normal user. Only through DUMP and of 32@, 32, and 32! the user can access the full 32-bits format (but this is done out of the Forth-79 Standard and is peculiar to qartus only).

See the paragraph "The memory model" in this manual.

[8] More from the inner help; type: qartus --help on the command line.

null

The value 0 (zero) marks the end of a string or the end of the input stream or the compiled code at the end of each definition in memory, to also mark the end of the execution tokens. Address 0 (zero) contains 0 (zero) and must not be changed, or the whole word-search engine will fail[9].

number

(definition by Robert L. Smith, with integrations of mine) A number is represented in the input stream as a word composed of a sequence of one or more digits, with a leading ASCII minus (−), if the number is negative, or a leading optional ASCII plus (+), if the number is positive, and at least an ASCII dot (`.`), in any position (but following the minus sign, in case of negative number), if the value is to be considered as a double precision number. A leading ASCII plus (+) is ignored.

Commas in the number sequence (at any position, but after the sign both for numbers and double length numbers) are ignored, and can be used for the three-digit division: `2,000,000.00`, which is far more comprehensible than, but equivalent to, the double length number `200000000`.

Examples of illegal numbers:

`- 345`	the leading minus is detached
`-34.5`	this is not a number (there's a dot)
`345-`	the minus is in a wrong position

Examples of illegal double numbers:

`.-345`	the leading minus is after the dot
`-345`	this is not a double number (a dot is missing)
`,-345.0`	the comma is before the sign

Numbers are classified according to the following table:

type	**range**	**minimum bit field**
bit	0..1	1
character	0..127	7
(unsigned) byte	0..255	8
number	−32,768..32,767	16
positive number	0..32,767	16
unsigned number	O..65,535	16
double number	−2,147,483,648..2,147,483,647	32
positive double number	0..2,147,483,647	32
unsigned double number	0..4,294,967,295	32

Numbers input follows these rules (expressed in `DECIMAL`), which adhere, *mutatis mutandis*, to the rules of the program R-Forth (the Forth interpreter used by Alan Winfield for his "*The Complete Forth*", cit.):

[9] Indeed, writing in this cell is prohibited with the usual methods. But devious programmers can conceive many methods to override the default methods, so I have to give this advice: don't change this value.

- a number in the range {-32768..32767} is pushed as-is (included bit, character and byte formats);
- a number in the range {0..65535} is interpreted as an unsigned integer and pushed as-is;
- a number in the range {-2147483648..2147483647} with at least one dot in any position (but after the minus in case it's negative) is interpreted as a signed double integer and pushed as-is.
- a number in the range {2147483648..4294967295} with at least one dot is interpreted as an unsigned double integer and pushed as-is.
- The position and number of the dots in the double numbers are not important, as long as they follow the minus sign in case of negative numbers. The variable `DPL` holds the number of digits after the *first* dot in the number; if the number is a single number, `DPL` is set to -1.
- a single number outside previous proper ranges evaluates arithmetically modulo 65536.
- a double number outside previous proper ranges evaluates arithmetically modulo 4294967296.

Warning: any bit configuration can be seen either as a signed or an unsigned number, both for single and double numbers, and the relative effects depend of course on the way the number is shown or used into calculations.

When represented in memory, numbers are contained in a cell, preceded by a code which specify the number type. The byte order within each 16-bit field is specified according to the rules defined in 'cell'.

When represented on the stack, the higher 16-bits (with sign) of a double number are most accessible. When in memory, the higher 16-bits are at the lower address. Storage extends over four bytes toward high memory.

`output, pictured`

The use of numeric output primitives, which convert numerical values into text characters sequences. The operators are used in a sequence which resembles a symbolic 'picture' of the desired text format. Conversion proceeds from lower to higher digit, from higher to lower memory.

`packed string`

See 'string'.

`return`

The means of terminating text from the input stream. A null (ASCII 0) indicates end of text in the input stream. This character is left by the 'return' key actuation of the operator's terminal, to mark a stop for the text interpretation.

`screen`

A memory space containing textual data (in Forth-79 usually called 'buffer'). By convention, a screen consists of 16 lines (numbered 0..15) of 64 characters (numbered 0..63), for a total of 1024 bytes (corresponding to 1 KiB or 1 block). Screens usually contain program source text. The first byte of a screen, once loaded, is the beginning point for text interpretation during a load. qartus has three independent screens with a dedicated memory space for each one.

`source`

Text consisting of word names suitable for execution by the text interpreter. Such text is usually contained in textual files or arranged in blocks maintained on a mass storage device.

`stack, control flow`

A last in, first out list which contains the machine addresses of decisional words, cycle and loop words during compilation, to pass reference addresses that must be compiled into the definition, and during interpretation for temporary values passed from one word to the other. The control flow stack does not belongs to qartus's memory and has a separate unreachable memory space of 9765 KiB of 32-bit numbers. This extra-wide control flow stack width was designed to guarantee the execution of programs written with recursive techniques which, when executed within structures, can cumulate a very high number of recursive steps. This value can be incremented by changing the constant `CFSTACKLENGTH` in '`tbas.h`', and by recompiling the sources.

`stack, data`

A last in, first out list consisting of cells. This stack is primarily used to hold intermediate values during execution of word definitions. Stack values may represent numbers, characters, addresses, boolean values, etc. The data stack belongs to qartus's memory. The data stack dimension can be resized with the console option `--stack-size`.
When the name 'stack' is used, it implies the data stack.

`stack, return`

A last in, first out list consisting of cells, which contain the machine addresses of loop word definitions whose execution must be transferred in case of cycles. The return stack may cautiously be used for other values, such as loop control parameters, and for pointers for text interpretation. The return stack belongs to qartus's memory. The return stack dimension can be resized with the console option `--return-stack-size`.

`string`

A sequence of 8-bit bytes containing ASCII characters, located in memory by an initial byte address, for at most 255 characters. When the string is represented by one value on the stack, the address, with the count on the first byte of the address, this string is defined as 'counted string' or 'packed string'. When a string is represented by two values on the stack, an address and a string count, the string is defined simply as 'string', or 'cell pair'.

A string is always case-sensitive.

If in a file the literal string is not terminated in the current line, it is automatically terminated by qartus. Strings thus do not extend over more lines.

`string, counted`

See 'string'.

`string, packed`

See 'string'.

`true`

A non-zero value representing the true condition flag. Any non-zero value will be accepted by a standard word as 'true'. All words returning a flag leave 1 as a 'true' flag.

`unit, memory`

See 'memory'.

`user area`

An area in memory which contains the storage for user variables and definitions. Usually, by default, the user area is about 51 KiB cells; if the user should use the options `--stack-size` and/or `--return-stack-size`, the stacks are incremented, and the user area is correspondingly reduced.

`vocabulary`

An ordered list of word definitions. Vocabulary lists are an advantage in differentiating dictionary search and in separating different word definitions that may carry the same name.

`word`

A sequence of characters terminated by at least one blank (or 'return'). Words are stored in input stream, from a terminal, files or mass storage device.

`word definition`

A named execution procedure compiled into the dictionary. Its execution is defined as a sequence of compilation addresses of other compiled words expressed in two bytes (plus any other byte necessary to complete the definition).

`word name`

The name of a word definition. Names are composed by any characters string up to 31 characters, and cannot contain ASCII nulls, blanks, or 'return'.[10]

[10] The Standard writes "..*Standard names must be distinguished by their length and first thirty-one characters*", suggesting that names might be longer, but only the first 31 make the difference. I found this memory-consuming, and practically inapplicable. So, at the cost of nothing, qartus stores only up to 31 characters; if name is longer, a condition of error may exist. So be careful, with long names.

1.4. Fulfilled requirements

qartus fulfills the requirements of the Forth-79 Standard.

The Forth-79 Standard recites: "*Each Standard System and Standard Program shall be accompanied by a statement of the minimum (byte) requirements for:*

1. *System dictionary space*
2. *Application dictionary space*
3. *Data stack*
4. *Return stack*
5. *Mass storage contiguous block quantity required*
6. *An operator's terminal.*

Each Standard System shall be provided with a statement of the system action upon each of the error conditions as identified in this Standard."

This is the data related to this fulfillment:

1. A 51 KiB system dictionary space (included in the dictionary)
2. More than 50000 bytes of memory for the applications dictionary
3. A data stack of 2048 bytes (that can be expanded)
4. A return stack of 2048 bytes (that can be expanded)
5. A mass storage capacity (65535 blocks for each directory)
6. A UNIX/Linux operator's terminal (direct or emulated).

For what about the fifth point, the mass storage is not necessarily contiguous, and its maintenance is accomplished by the Operating System into a user-chosen directory, where blocks are textual files numbered 1 to 65535. There may be any number of such a directory, that the user may create anew. This is a minor deviation (an enhancement, actually) from the Standard requirements.

1.5. The memory model

qartus runs in a single, contiguous, flat 32-bit address space. Each 32-bit address hosts the single byte of one Forth memory position; special code characters (which are 32-bit numbers) are hosted in the same 32-bit address, but this is transparent to the user.

qartus's memory is organized, as the Standard of Forth-79 requires, in bytes (i.e. an 8-bit unit); a byte or a character is stored in one byte; a (single) number is stored in two bytes (i.e. a 16-bit number, named 'cell' in Forth-79 jargon); and, as expected, a double number is organized in four bytes (i.e. a 32-bit number, thus in two cells). Cells are the basic number format, the one you see in the stack; the single 8-bit units can be accessed from memory with the Standard words `C@` and `C!` (and `C,` which I added for your ease).

As said, the byte, an 8-bit unit, is hosted in a 32-bit address, capable of holding more than 8 bits; the system uses the full 32-bits for control codes, markers, internal addresses and for all that is necessary for compilation; these extra bits are useless for the normal user and normally inaccessible (unless the elective word `32@` is used, but this is not a Standard word).

The memory model is organized in sections, as in fig. #1, where higher addresses are on top.

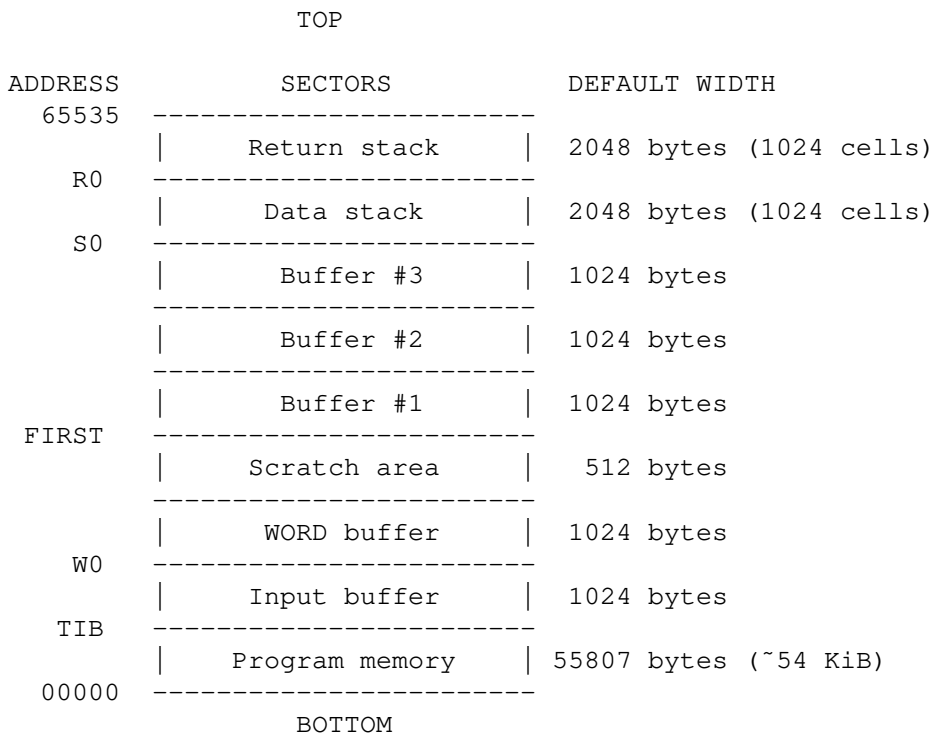


Figure #1

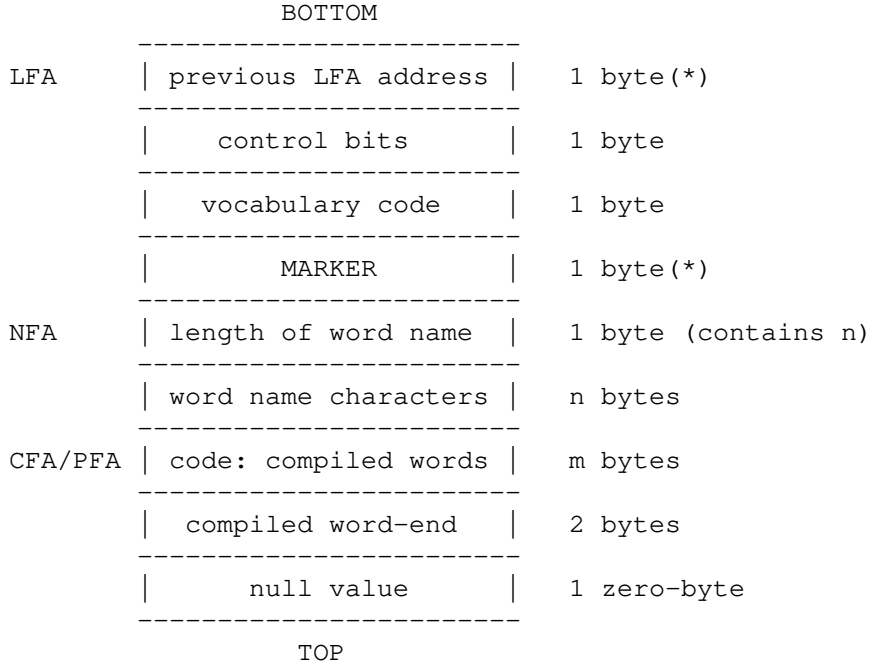
The labels on the left identify the constants in the dictionary to use to address specific parts of qartus's memory. This memory, which is set to 65536 bytes ranging from zero to 65535, cannot be expanded (because the unsigned single number has range {0..65535}). Let's review the various memory sectors:

- The dictionary has a total capacity of circa 54 KiB (55807 bytes). All compiled words, variables, constants and the `ALLOT`ed space is stored in the dictionary. The built-in words consist of about 3 KiB, so the user area is about 51 KiB. If 51 KiB seem poor for your programs, remember that each definition is stored in a few bytes.
- The input buffer (a.k.a. Terminal Input Buffer) has a total capacity of 1024 bytes and hosts the current input line (from the keyboard or a block); the system uses this buffer to evaluate the input from the keyboard, file or block.
- The `WORD` buffer is dedicated space storage of 1024 bytes that `WORD` uses to store its data; this buffer is split into four segments of 256 bytes each, available in turn, and the current segment address is returned and can be used and copied anywhere. I set the `WORD` buffer apart from the program memory, to avoid overwriting of any kind. The system constant `W0` identifies the first byte of the first segment of the `WORD` buffer.
- The scratch area is used by the pictured output strings created between `<#` and `#>` and is also a dedicated buffer. It's 512 bytes wide, sufficient for any output format.
- Numbered buffers #1, #2 and #3 refer to the screen buffers where `BLOCK` loads the correspondent storage blocks in chunks of 1024 bytes. The system constant `FIRST` identifies the first byte of screen buffer #1.
- Finally, in the higher part of the memory system, stacks are found: the Data Stack and the Return Stack, accessible by the qartus programmer using the usual stack words, both 2048 bytes wide (corresponding to 1024 cells); System constants `S0` and `R0` identify the first byte of the Data Stack and the Return Stack respectively; there is a third independent stack, a 256K 32-bits numbers stack, only for system usage, destined to internal passages (mainly control flow and decisional steps taken during

the compilation) and normally inaccessible.[11]

You can use the buffers of WORD and the Scratch Area as a contiguous 1536 bytes private buffer, for your own scopes (provided you don't use WORD and the pictured formats). To retrieve the proper addresses, use INFO.

Each word (user or built-in) is saved in memory, either by the colon/semicolon mechanism or by any defining word as CREATE or VARIABLE, in a fixed scheme (as in fig. #2, where cells have been classified according to the fig-Forth jargon - I know this scheme is not Standard).



(*) in full 32 bits.

Figure #2

The *Link Field Address* is the address where the word begins, that contains the link to the previous word for backward search; it is also the word address stored in memory when that word is compiled; it's noteworthy underlining here that this address is stored in full 32-bits, for speed reasons.

Control bits and the vocabulary code establish the word functioning. The MARKER, a 32-bit number, serves as a known point for traversing words upward and downward. There is a limit of 29 vocabularies you can create (they are 31 in total, with predefined unchangeable vocabularies #0 = FORTH and #1 = EDITOR).

The *Name Field Address* is the address where the name characters begin. Its first address contains the word count. There is a limit of 31 characters for the length of the names you can use.[12]

The *Code Field Address* is the start of the compiled code in words, and thus I suppose it matches the

[11] You can access the Control Flow Stack using CF-GET, which removes the top of the Control Flow Stack and pushes it on to the Data Stack and CF-PUT, which pushes the top of the Data Stack on the top of the Control Flow Stack; they may simulate the compilation process of words that require passing data thorough the Control Flow Stack; you can also print the content of the Control Flow Stack with the service word CF@. See the chapter "*Service words.*"

[12] This limit is established to 31 by default because it is required by the Standard. You can, anyway, change this value in custom.h, setting a different value. This value should not be lower than 31, because this could cause a system crash at the start or while loading items from qlibrary.ft. See also WIDTH.

definition of 'Compilation Address' reported in the Standard; `FIND` returns the Code Field Address, and `EXECUTE` executes the Code Field Address.

The *Parameter Field Address* is the start of the parameters (i.e. values) that some items do contain. It matches quite always the *Code Field Address*, but not for all items. Read the paragraph "*A word about* `FIND` *and ' (tick)*" for more. Read also the Appendix for information about qartus internals.

The compiled word-end appears only for colon words, vocabularies and `CREATE`d words, and not for variables and constants.

Strings are usually stored in memory as counted strings, i.e. the first byte holds the length L of the string (up to 255 characters for each string) and it's followed by the L characters that make up the string, one byte for each. If a string terminates before its counted limit (for instance it may be typed with `EXPECT`), it is closed with the NULL character. Any typing word prints thus the string until its natural end (the characters count) or until a NULL is encountered. This is a minor deviation (an enhancement, actually) from the Standard.

1.6. The run-time engine

Let's review some aspects of the run-time engine.

1.6.1. The execution model

As reported also in the man page, the interpreter, before starting the Forth session (the idle cycle), executes the following processes, in order:

1 Populate the dictionary with built-ins and sets up system variables.

2 Process the `--init` option. (if given)

3 Parse the resource file `$HOME/.qartus.rc`. (if any)

4 Parse the files in the argument list. (if any)

5 Process the `--command` option. (if given)

The resource file is searched in `$HOME`, and its name is `.qartus.rc`[13]. This file (which is a Forth source, after all) is not created by qartus. If you want to permanently activate some options, compile words, create variables, populate the stack, print messages and banners or whatever, create the resource file by yourself and, from now on, the program/data in the resource file will be executed and be a permanent part of your system. If you remove the resource file, qartus won't complain and will start a normal default session.

1.6.2. The Program Counter

When an instruction is under execution, its address is stored in the Program Counter, which is a pointer which moves through the dictionary to execute program instructions, even in case of nested words. This means that the 'next instruction code' is not pushed to the Return Stack (so common for fig-Forth and R-Forth users), but it's held in a system variable, called `PC`[14] (Program Counter), that can be altered to modify the control flow; this process avoids dangerous usages of the Return Stack (apart from `DO..LOOP` and `LEAVE`, which use the Return Stack for saving the intermediate limit and index, available for the word `I` to retrieve the current index value).

[13] Under the Windows Prompt, the file is renamed `_qartus.rc` and is not hidden. So Windows users may take care of this because they don't get the same effects running under Cygwin or under the Windows Prompt.

[14] In fig-Forth it was know as `IP`, that is the *Interpretive Pointer*.

If you plan to use PC, remember though that the engine of qartus updates PC after each instruction by adding two (one for the instruction id, one for the instruction code); besides, you must pay attention while altering PC, because some words (e.g. IF, ELSE) have three cells, and others (e.g. ."") use a variable-length cells number.

During the debugging, PC is shown at each step.

1.6.3. The 'readline' facility

qartus is built around the 'GNU readline' library facility, that lets you type a few letters of the dictionary word[15] you want to invoke and, by hitting the TAB key, either readline completes the word (if there is one only choice) or it prints a set of possible alternatives. Typing one or two more characters, in general, shortens the possibilities of finding rapidly what you're looking for.

This makes typing faster, and long words are not a problem anymore.

One feature that I appreciate is the completion facility, which eases in typing words or file names. The facility by default starts with dictionary-word completion; the second completion is started by putting a "here-file" character (in UNIX jargon, it means using ./"): this instructs qartus to step to the file-completion rather than to the dictionary-words completion, which is restored at the ENTER key. This is activated only from the second word on (that is only if at least one space is found in the input line); this assures that any first word is always searched in the dictionary. This is extremely useful with REQUIRE, ENSURE or INCLUDE, just to tell the more common examples.

Readline is common in all UNIX shells, because of its versatility; it is common in gforth too. Thanks to the guys at the Free Software Foundation project. (You can find readline news for example at https://tiswww.case.edu/php/chet/readline/rltop.html.)

1.6.4. Program interruption

Sometimes, you might find that your program hangs, due to one of the following cases:

a) because of a compilation error from a file
b) because of some calculation that finds wrong data on the stack
c) because you wrote accidentally an infinite cycle
d) because the system has become unable to accept more characters
e) because you accidentally overwrote some dictionary part

then do one of the following actions:

- Press CTRL-C and then press ENTER a few times until you see ok.
- type BYE to end your session, or
- type COLD to reload the built-in dictionary (losing all changes so far)
- If the error was in a compilation state, and this state is still active, type ; or [to re-enter the interpretation state and proceed.
- If you should find yourself in the 'comment' mode initiated by (, type) to re-enter the interpretation state and proceed.
- If anything fails but the system is still responsive, try with CHECK-DICT, which scans the dictionary memory from the bottom and tries to remove the words that make the system unresponsive. If even this should fail, a COLD or BYE must be invoked.

[15] Super-immediate words are not detected by readline, so that the word completion is not enabled for them.

If all previous tips don't work for you, write to me. The problem may be in qartus's internals. Follow the instructions in the man page.

1.7. System constants

qartus's dictionary contains some system constants that help the programmer evaluating the environment in which his programs are run. They are 16-bit constants (the predefined value, if already known, is reported in square brackets):

B/BUF, the dimension of a screen buffer in bytes [1024].
B/SCR, the number of screens per block [1].
BL, the ASCII code for the blank space [32].
C/L, the number of characters per line [64] (numbered 0..63).
ERNUM, the number of default error messages in current version [61].
FALSE, the false value 0
FIRST, the address of the first screen buffer character. (*)
L/SCR, the number of lines per screen [16] (numbered 0..15).
LIMIT, the address just above the highest screen buffer character (*)
NOOP, the code for the compiled no-op function. (*)
OFF, the value 0
ON, the value 1
R/O, the Read file access method code [1].
R0, the start address of the Return Stack. (*)
S0, the start address of the Data Stack. (*)
STDERR, the standard error channel code (use with WRITE-LINE)
STDIN, the standard input channel code (use with READ-LINE)
STDOUT, the standard output channel code (use with WRITE-LINE)
TIB, the start address of the Terminal Input Buffer. (*)
TRUE, the true value 1
W/O, the Write file access method code [2].
W0, the start address of the WORD buffer. (*)

(*) Value calculated at start.

1.8. System variables

qartus's dictionary has some system variables that help to keep track of the dictionary status or to govern some processes. They are all 16-bit variables (the proper usable range is written in curly brackets):

#BUFFERS, the current number of active screen buffers {0..3}
#CLOCK, the milliseconds/microseconds index for CLOCK {0, non-zero}
#L, the current editing line in current screen {0..15}
#P, the current editing position in line in current screen#L {0..63}
#TIB, the current character count in TIB {1..1024}[16]
#VOCS, the current number of vocabularies in the system {2..31}
>IN, the current position in TIB {0..1023}
BASE, the current number base (default 10) {2..36}
BFR, the current screen number being EDITed or LOADed {1..3}
BLK, the current block number being loaded {0..65535}
CONTEXT, the current search vocabulary reference {0..30}
CSP, a user variable containing a stack index {0..65535}
CURRENT, the current dictionary number where new definitions are created {0..30}
DEBUG-STATE, the debug state flag {0, non-zero}
DPL, the number of digits typed after a dot in a double number {0..255}

ERR, the last error code set by ERROR {-32768..32767}
FENCE, the address under which FORGET cannot actually forget {0..65535}
FIG-STATE, the fig-Forth compatibility flag {0, non-zero}[17]
FLD, the number of characters in the last pictured output {0..255}
HLD, the address of last characters in the last pictured output {0..65535}
IPx, the containers for the Internet Protocol 4 (x = 1..4)
LAST, the address of the last initiated dictionary entry {0..65535}
OFFSET, the offset from the address returned by BLOCK {0..1023}
OUT, the current printing position counter {0..255}
PC, the current program counter {0..65535}
R#, the current cursor offset in the entire block screen {0..1023}[18]
SCR, the current editing block number {0..65535}
SPAN, the characters count after last EXPECT {0..255}
STATE, the current state {0 or 1}
WARNING, the current mode of the ERROR management {1,0,-1}
WIDTH, the maximum length for the definition of words {1..255}

These variables have the limits reported in parentheses. Variables that have a range of 8 bit ({-127..128} or {0..255}) use only the low part of the two bytes; the other variables use the both of the bytes, that is the whole cell. When variables have a range {0..65535} they must be considered unsigned.[19]

1.9. Error management

The Forth-79 Standard specifies that an error found during an execution process must be taken into account by the Forth interpreter with a proper message, without specifying *how* this must be accomplished. So I grabbed some ideas from the fig-Forth (an environment with a very specialized set of dedicated words), and wrote all the error messages with the purpose to have them clear and complete.

1.9.1. Default Error Management

When an error condition exists, control is passed to a system error procedure, along with an error code; the system error procedure then prints the system error string corresponding to the error code.

The system proceeds now to solve the error condition by:

- setting STATE to interpret mode
- resetting all stacks
- resetting HERE in case of an error in compilation
- finally, control is returned to the keyboard.

---

[16] The two variables >IN and #TIB refer to the same Input Buffer, with this important difference: >IN has conventionally range {0..1023}, and it is the pointer to the Input Buffer used by the text interpreter, incremented at each word invocation (starting from zero); #TIB instead is updated by the system only when the Input Buffer is filled and reports the total number of characters in the Input Buffer. After the commands in the Input Buffer have been completely evaluated, the two variables point at the same character, but #TIB has **always** pointed to it, while >IN has progressively reached that point; besides, >IN is one unit less than #TIB, because >IN is a pointer, while #TIB is a quantity.

[17] The FIG-STATE compatibility flag holds the forth state, zero for Forth-79, non-zero for enabling the fig-Forth features (described further in the manual).

[18] This variable holds actually the result of: #L * 64 + #P.

[19] It's curious noting that the Standard of Forth-79 got rid of most variables that the fig-Forth openly created and used; I found this someway weird: what harm can a variable ever do? Probably the 'STATE variable' affair may have influenced the decision. (A lot of critics wrote about STATE and the damage it can do.) I feel, instead, that system variables can increase the knowledge of the current and run-time state of the Forth engine; after all, if you don't want to use a variable, ignore it! But if you need it, here it is!

These are called the "default error resolution operations."

This Default Error Management manages all cases and should be sufficient for all needs.

1.9.2. User Error management

If you wish to print your error messages or you want to execute different routines to resolve the error, before the default error resolution operations, the User Error Management will help. This special management can be used through the following built-in words:

The word ERROR is used to enable the User Error Management. The argument of ERROR is a number corresponding to the index of a string in the default errors string array, or a line in a screen buffer, or a user code in an alternative user procedure.

The user variable WARNING rules the user error management behaviour: it contains 1, 0 or -1.

> When WARNING is zero (default state), ERROR uses the argument as an error code and calls the Default Error Management; at all effects, it's like a system error procedure performed by qartus. You can use the system error strings to flag errors likewise the error system does on its own.
>
> When WARNING is 1 and a screen was at least loaded (that is, BFR is not null), ERROR retrieves the output strings from the active screen buffer, with errors codes in the range {0..15} (these strings may also be simple messages and not necessarily error messages, though passed through ERROR). BFR can be used to set/unset the current screen buffer and #BUFFERS can be used to check if at least one screen was loaded. Screen 20999 offers an example of an error messages screen.
>
> When WARNING is set to -1, ERROR calls the word (ABORT); this is an existing run-time procedure that in its default state simply calls ABORT, but the user is free to implement his own (ABORT) word and this last will be called by ERROR. In the user redefinition of (ABORT), the user may read ERR to know the last used error code and then add his messages, variables resetting, all that's needed to resolve the user error condition (for instance illegal values in a math function).

After returning from ERROR (either case), the default error resolution operations take place, before returning control to the keyboard.

The word ?ERROR calls ERROR if the flag found on top of the stack is true (conditioned error simulation).

The word MESSAGE simply prints the error message corresponding to the error code found on the stack (depending, like ERROR, on WARNING), and no default error resolution operations are performed. MESSAGE can be used to test your error routines, without the need of causing a real error.

The variable ERR contains the last error code. This variable is set by ERROR and should only be read, not written, by your (ABORT) procedure.

You can get the system error list are by typing

```
$ qartus --errors-list
```

from the console (outside qartus's environment).

1.10. Using BLOCK and BUFFER

The default Forth-79 way to store information (Forth was not a file-driven environment) is the blocks set. A block is a contiguous space of 1024 bytes (characters) capable of storing anything in textual form (programs, ideas, your pasta recipe, your girlfriends' cellphone numbers, texts and even books).

The 1024 quantity was chosen for three reasons: the first is that it is a power of two, and this has some fascination; the second is that it is the power of two closer to 1000, which is another fascinating number; the third is that it can be split on a screen and shown (or edited) in 16 lines of 64 characters, which seems a very reasonable screen space for humans.

1.10.1. Using BLOCK

BLOCK needs a block identifier (an unsigned integer un) on top of the stack. If un is not an available block number (the only case in qartus is if un = 0) an error is generated. The returned value is addr, the address of the first character of the block screen assigned to mass-storage block un.

If block un is already in a block screen, addr is the address of that block screen.

If block un is not already in memory and there is an unassigned block screen, BLOCK transfers block un from mass storage to an unassigned block screen, and addr is the address of that block screen.

If block un is not already in memory and there are no unassigned block screens, BLOCK assigns the least accessed block screen. If the block in that screen has been UPDATEd, it transfers the block to mass storage and transfers block un from mass storage into that screen, and addr is the address of that block screen.

After the operation, the block screen pointed to by addr is the current block screen and is assigned to un.

1.10.2. Using BUFFER

BUFFER needs a block identifier (an unsigned integer un) on top of the stack. If un is not an available block number (the only case in qartus is if un = 0) an error is generated. The returned value is addr, the address of the first character of the block screen assigned to mass-storage block un.

If block un is already in a block screen, addr is the address of that block screen.

If block un is not already in memory and there is an unassigned block screen, BUFFER sets addr as the address of that block screen.

If block un is not already in memory and there are no unassigned block screens, BUFFER assigns the least accessed block screen. If the block in that screen has been UPDATEd, it transfers the block to mass storage, and addr is the address of that block screen.

After the operation, the block screen pointed to by addr is the current block screen and is assigned to un.

In the gforth manual, there is a hint that is also valid for qartus (and for other fig-Forth and Forth-79 systems too), that suggests that the subtle difference between BUFFER and BLOCK is that you should only use BUFFER if you don't care about the previous contents of block un.

A statistic based on *Forth Dimensions* (made informally by me) shows that BUFFER is seldom used, if never. But it's a Standard word anyway.

1.10.3. Choosing specific blocks directories with USE

The system blocks are by default contained in $HOME/.qartus/blocks/, which is created by the first program execution and populated by the --config option (see the file INSTALL); if you want to use another directory, you can type

```
USE mydir
```

If the name is a relative one, the directory mydir will be created in the current directory (the one where you are) if not existing. You can specify also an absolute path for it:

```
USE /home/user/data/qartus/proj1
```

All new blocks created with EDIT or loaded with LOAD or searched through SEARCH will use the specified directory from now on. The other directories will maintain their contents so that you can create one directory for each project (or group of projects) and simply pass from one to the other by calling USE on each.

Calling USE alone, without arguments, simply resets the system directory $HOME/.qartus/blocks/:

```
USE
```

1.11. Extensions for File-treatment

UNIX® is an Operating System dedicated to files. It would be strange programming under UNIX without accessing any kind of (textual) file located anywhere in the file system, and not only in BLOCK files. Since Forth is definitely not a UNIX-file-oriented language, I introduced the following low-level non-Forth-79 procedures:

```
OPEN-FILE
CLOSE-FILE
READ-LINE
WRITE-LINE
```

and the following procedure to return a file access method:

```
R/O
W/O
```

(in all they are a subset of the ANSI 1994 File Support Words) to give the programmer a bunch of tools useful for accessing files on a UNIX Operating System.

OPEN-FILE needs a string address and a string count (which may be retrieved through EXPECT or use some trick like S" contained in the library), and a file access method. The file name must reside on the file system, of course, and the path (if necessary) should be contained in the file name.

If you open a file in reading mode (using R/O), the file must exist. The file pointer is positioned to the beginning of the file for reading the file content from the start. Reading is done through READ-LINE, which needs a string address and a string count to store the line read (it suffices a CREATEd and some ALLOTed space), but it also needs the file-id that OPEN-FILE returns in case of a successful opening. I suggest saving this file-id (which is an integer) into a variable, avoiding some stack noise.

Opening a file in write mode (using W/O) causes the file to be erased if existing or created empty if not existing. Opening a file in append mode (using R/O + W/O) needs the file to exist. The file pointer is positioned at the end of the file so that subsequent writing adds contents to a file (appends). Writing, in either

case, is done through WRITE-LINE which, as for READ-LINE, needs a string address, a string count and a file-id.

CLOSE-FILE does not take into account the opening mode: it simply ceases the connection with the file, saving information permanently.

These words to me seem sufficient to open a file and read lines; it's up to the programmer separate the content of the lines and use the information therein contained.

1.12. Using computer ports

It would be strange for a forth interpreter being unable to talk to external devices. It would be a blind and self-circuited environment. So I enabled qartus to be aware (at least potentially) of the computer ports. It is the responsibility of the programmer talking to the right port for the right scope but, if this is intended, the ports talk is easy.

Warning: if the port words do not return the desired values, perhaps they are under control of the firewall, which could prevent the use of some of them. So check it out!

The connection is opened with a talk to 127.0.0.1 by default (that is to your computer ports). If you need another address, change IP1, IP2, IP3 and IP4 (four variables of the default system), which define the address to open. At start IP1=127, IP2=IP3=0, and IP4=1. These are the characteristics of the connection:

- The connection type is SOCK_STREAM: it provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.
- The protocol is AF_INET, that is it uses the Internet Protocol 4 (IP4).
- The reading or sending proceeds by bytes (one byte for each reading).

To enable the connection to a port, the PORT word must be used, in connection with the constants ON e OFF, to enable/disable the port connection. The port number is expected on the stack:

```
un ON PORT
```

will try to open and connect port number un, while

```
un OFF PORT
```

will close the connection with port un. At most 64 contemporary connections can be established by the PORT word.

OFF PORT does not return any status value.

To receive and send bytes from/to the port, the words LISTEN and SEND may be used.

LISTEN needs the port number on top of the stack, and returns the received byte. a value of 0 means an empty receiving or an error in reading. A positive value means correct receiving[20]. E.g.

```
13 LISTEN
```

SEND also needs the port number on top of the stack, and the value to send underneath. For instance, to send the letter 'A' (65 in ASCII) to port 13:

```
65 13 SEND
```

This site offers a list of all ports of a common PC:

```
https://www.hostingreviewbox.com/rhel-tcp-and-udp-ports/
```

1.13. Making sounds with qartus

The dictionary of qartus includes a couple of words for producing sounds. The first is `BELL`, which was also in the Forth-79 dictionary, though only in the Reference Word Set, probably due to the different computer architectures at those times, some with sounds capabilities and some without. The second is `(SOUND)`, which produces a tone variable in duration and pitch. Let's review both of them in details.

1.13.1. Using `BELL`

`BELL` is available (with identical purposes and effects) as `BEEP` (guaranteeing compatibility with other Forth-79 or fig-Forth implementations.

In the `qartus.h` header file there are five definitions that are useful to choose the right method for the current architecture: the `SPEAKER1` to use the Sox software, the `SPEAKER2` to use the speaker-test software and the third to use the ASCII Bell System. There is also a `SPEAKER4` version for doing nothing under Linux, Unix or Cygwin and `SPEAKER5` for doing nothing under the Windows Prompt[21]. The general variable `SPEAKER` is defined in `custom.h` and set to one of `SPEAKER1`, `SPEAKER2`, `SPEAKER3`, `SPEAKER4` (on Linux/Unix/Cygwin) or `SPEAKER5` (on Windows only) before compiling (default is `SPEAKER1`).

1.13.2. Using `(SOUND)`

The second versatile sound word is `(SOUND)`, which is capable of producing a sinusoidal tone variable in duration and frequency. This is an experimental word that is not really portable on all Operating Systems. So use it with care.

Again, in the `qartus.h` header file there are four definitions that are useful to choose the right method for the current architecture: the `SOUND1` to use the Sox software, the `SOUND2` to use the speaker-test software. There is also a `SOUND3` version for doing nothing under Linux, Unix or Cygwin and `SOUND4` for doing nothing under the Windows Prompt[22]. The general variable `SOUND` is defined in `custom.h` and set to one of `SOUND1`, `SOUND2`, `SOUND3` (on Linux/Unix/Cygwin) or `SOUND4` (on Windows only) before compiling. (Default is `SOUND1`.)

The word `(SOUND)` uses two values on the stack, the topmost is the frequency in Hertz and the second the duration in milliseconds. (1000 is one second.)

Use this feature with care, because it's still experimental. In particular, the sound at present blocks the execution of the program, and so, if you create a piece of music to be performed by your program, the program won't proceed until the entire music is over.

---

[20] Warning: sometimes the port does not return zero but the last available code repeatedly (generally the value 10 for `\r` or 13 for `\n` from ports that don't end transmission, like port 13); this is not an error in transmission, so, in these cases, you should check not only for zero but for 10 or 13 also, or stop reading if a non-printing character is met. See for instance the word `GETDATE` defined in the library.

[21] This is the case for producing no nasty sound or graphical effects for `BELL` or `BEEP`, under Linux, Unix or Cygwin, or even under the Windows© Prompt.

[22] This is the case for producing no nasty sound or graphical effects for `(SOUND)`, under Linux, Unix or Cygwin, or even under the Windows© Prompt.

1.14. Pipes with qartus

Piping into qartus is possible through the reading with `READ-LINE` from `STDIN`, and output to a pipe can be done with `TYPE`, `EMIT`, `CR` and the like. Here's a detailed exposition with pipes[23].

1.14.1. Reading a pipe

The reading from a pipe is done using `READ-LINE` repeatedly from `STDIN`; the end of the pipe is signaled with the return value 0 as the second value on stack, which is 1 for a correct reading and 0 for an error or the end-of-pipe signalling. If you want to be more specific (for example testing the difference between the end-of-pipe and the failing reading from a pipe) you should also check for the first value on stack - the `ior` in ANSI 1994 jargon - which is 0 for a correct execution, 1 if the pipe was never opened, and 2 in case of errors in transmissions (see `READ-LINE`).

Here's an example of an invocation of qartus usable in a pipe:

```
 qartus --init=": foo begin pad dup 84 stdin read-line drop while type
repeat ; foo bye"
```

This example works like the UNIX shell tool `cat`, to copy the pipe and type it on the screen. An example of a real usage of such a piping is[24]:

```
 $ cat fibo.ft | qartus --init=": foo begin pad dup 84 stdin read-line
drop while type repeat ; foo bye"
```

1.14.2. Writing to a pipe

As said earlier, the writing to a pipe is performed simply by using the printing words that output ASCII text to the screen, like `TYPE`, `CR`, `EMIT`, the dot family and the like.

An example of an output to a pipe is a reworking of the previous example, that reads from a pipe, and passes the output to the UNIX shell tool `head` to print the first 10 lines.

```
 $ cat fibo.ft | qartus --init=": foo begin pad dup 84 stdin read-line
drop while type repeat ; foo bye" | head
```

This chunk uses `READ-LINE` to read *from* a pipe and `TYPE` to output *to* a pipe, intercepted by `head`. As a UNIX program, qartus behaves naturally with pipes, which are a UNIX essential feature.

1.15. Super-immediate words

Some words, due to the particularly delicate role they play in the run-time architecture of qartus, were marked as *super-immediate*. This means that they are intercepted before the interpreter. Since they therefore don't need to be in the dictionary, they can be used even in case the dictionary is messed-up, or in case the system gets unstable due to some strange experiments (I speak by experience). Two of these words are `BYE` and `COLD`.

The super-immediate words cannot be compiled, not even with `COMPILE` or `[COMPILE]` (with the exception of `BYE`, for its wide historical importance). They act always, immediately, and need no stack values or buffers to operate. They are nonetheless described in the manual, because of their utility. Their usage and specifics are reported in the chapter "Service words".

---

[23] I derived much of this from the gforth manual.

[24] Gforth creators: forgive me if I choose your exact name for this procedure, but you set someway a definitive word in this subject, so I simply retrace your steps...

2. Some discussions about Forth-79

In the following I present some discussions about some Forth-79 requirements and how qartus behaves, with all the considerations that lie behind the design. I thank Robert L. Smith for his illuminating articles about some Forth peculiarity, that were useful during the design phase of qartus.

2.1. A word about FIND and ' (tick)

The two words FIND and ' (tick) have in Forth-79 similar but subtly different scopes. While FIND retrieves, from the input stream, the Compilation Address of a word (in qartus's jargon this means the Code Field Address), that can be conveniently used by EXECUTE in a vectored execution, the word ' (tick) retrieves from the program text the Parameter Field Address, and thus can be conveniently used to inspect a variable's content or a word's compiled code.[25]

In qartus's dictionary, the CFA and PFA coincide only for colon words and vocabularies; for all variables, constants and CREATEd entities, CFA and PFA addresses differ. To be sure to use the correct data, use always ' for PFA and FIND for CFA, even if they might return the same value.

The words CFA, LFA, PFA, NFA (which can be found in the qlibrary.ft file), belonging to the fig-Forth tradition, can be ENSUREd and used to convert one address to another, in particular:

CFA converts a PFA to CFA (may result, as said, a no-op)
LFA converts a PFA to LFA
NFA converts a PFA to NFA
PFA converts an NFA to PFA

Read also the interesting article by Robert L. Smith, titled "*Compilation Addresses and Parameter Fields*" in "*Forth Dimensions*" Vol. IV, N. 6 March/April 1983.

2.2. DO, LOOP and +LOOP problems

In his article "*Forth Standard Corner - DO, LOOP, and +LOOP*", on *Forth Dimensions* vol. III N. 6, Robert L. Smith describes some problems that arise in the Forth-79 Standard definition of the DO..LOOP and DO..+LOOP cycles.

The first problem deals with memory addressing; since DO cycles use signed values as indices, when these indices are used as memory slots, the memory addressing is not guaranteed, because in the Standard there's no requirement about this. qartus, in dealing with addresses, uses only unsigned references, so that the indices in the DO cycles are implicitly converted to unsigned when treated as addresses, but the problem posed by Mr. Smith is a serious one, and in following versions of Forth the DO..LOOP cycles were defined to treat unsigned values.

In a 64 KiB Forth-79, to traverse the whole memory, the solution is writing the following:

```
32767 0 DO .. LOOP (from 0 to 32766)
... 32767 ... (the 32767 case)
0 -32768 DO .. LOOP (from 32768 to 65535).[26]
```

The only caution is that the address location 32767 is unreachable with these cycles, so it's necessary to

[25] The Forth-83 and later changed these things: the Forth-79 ' (tick) was renamed ['] (bracket-tick) and constrained to compilation only, to retrieve the word name from the program text; the ' (tick) was retained to retrieve the name from input stream (like Forth-79 FIND). FIND was changed to retrieve the word name from a counted string. In Forth-83 and later, ', ['] and FIND all return an execution token (that is a Code Address). The word >BODY was added to convert an execution token to the start of the parameter field.
Well, I think Forth-79 had a neater design. My opinion, though.

treat this value singularly after the first cycle.

The second problem posed by Mr. Smith is when the `+LOOP` increment is zero, that is `n=0`; this is a fake problem.

The solution in the Standard is to end the cycle when "*...the new index is equal to or greater than the limit (n>0)* [writing n>0 instead of n≥0]*, or until the new index is less than the limit (n<0)*". The treating of the limit poses two alternatives:

1) the cycle never ends.
2) the cycle stops after the first turn.

For qartus I chose to implement the second solution, because this avoids the infinite cycles (always dangerous). Thus, the case of:

```
11 0 DO .. 0 +LOOP
```

makes the cycle stop after one turn (and this anyway obeys to the first condition in the Standard).

The chosen solution also obeys to the first and second conditions in the Standard, and rules the cases of increments with sign against the normal cycling versus; for instance:

`11 0 DO .. −2 +LOOP` (supposed to run from 0 to 10)

or

`0 11 DO .. 2 +LOOP` (supposed to run from 11 to 1)

The first case, with `n<0`, stops after one turn, because after the first cycle, the new updated index (0-2=-2) is actually "*less than the limit*", which is 11. The second case, with `n>0`, stops equally after the first cycle, because the new index (11+2=13) is "*equal to or greater than the limit*", which is 0. This is how qartus behaves in such cases.

So the problem seen by Mr. Smith was solved by choosing the second alternative.

One final consideration: the then-to-come Forth-83 standard will require that "*...The loop is always executed at least once. For example:* `w DUP DO ... LOOP` *executes 65,536 times...*". In Forth-79 (and in qartus), instead, since values are unsigned, the cycle is executed *only* once:

```
n DUP DO .. LOOP
```

After the first cycle, `n` is incremented to `n+1`, and since it is greater than or equal to the limit (which is still `n`), the cycle stops. The same, with analogue reasoning, is applied to `+LOOP`. There is not an easy way, in Forth-79, to run one single cycle 65536 times. The highest run count is actually 65535:

```
32767 −32768 DO I 32768 + U. LOOP
```

This program token prints all numbers from 0 (unsigned) to 65534 (unsigned), in total 65535 numbers. The loop limit is always excluded and must be considered apart.

[26] The cycle traverses all numbers from -32768 to -1, and of course, -32768 is the unsigned 32768 and -1 is the unsigned 65535.

2.3. Something about EXECUTE

Actually, the definition of EXECUTE in the Standard is "*Execute the dictionary entry whose compilation address is on the stack.*"; this definition is indeed not complete, and does not clarify the concept of 'what' is to be executed, that is it does not explain if the compilation routine executes the single address argument or this is considered the start of a procedure traversing all addresses from here until an EXIT or an end-of-word token is met.

2.3.1. The 'what' side: Execution of the compilation address

The fact it compiles only compilation addresses is here explained. Take a look at the following examples:

A compiled address, say 4290, contains 230 53 (it corresponds to the built-in . - the dot - where 230 is the builtin execution token and 53 the code for dot, the xt in ANSI jargon); if this is case, the calling

```
24 4290 EXECUTE
```

would print 24 (it is equivalent to 24 .).

But if in one memory address, say 6780, we store the values 194 16 (which is the address 4290), the calling

```
24 6780 @ EXECUTE
```

would also print 24; the address in 6780 is evaluated, and searched, founding a builtin word.

On the contrary, the phrase

```
24 6780 EXECUTE
```

would not print 24, because the address does not contain a compiled region, but a numeric address.

2.3.2. The 'how many' side: Multiple or single execution

The second concept is hidden in the definition. A doubt arose in me while reading the definition of EXECUTE given by the ANSI 1994 Standard: "*Remove* xt *from the stack and perform the semantics identified by it. Other stack effects are due to the word* EXECUTE*d.*". This definition suggested to me that only the xt on the stack was performed. But this simple example performed in a gforth sessions showed it's not true:

```
: attempt ." First" cr ." Second" cr ;  ok
attempt First
Second
 ok
variable xt ' attempt xt !  ok
xt @ execute First
Second
 ok
```

The variable xt contains the compilation address of attempt, which has four builtin words compiled in it. EXECUTEing this compilation address executes in turn all the builtin words compiled into attempt.

If you re-read carefully the Forth-79 Standard, you may notice the expression used is "entry", which implies that it is a starting point, not a single expression.

In conclusion, I think that for what about EXECUTE, the Forth-79 Standard definition is better than the ANSI 1994 Standard definition.

2.4. `LEAVE` or not leave?

In his article "Forth-79 Compatible LEAVE for FORTH-83 DO..LOOPS", Klaxon Suralis writes that the `LEAVE` in Forth-79 is (I quote) the "*horror of horrors*" because `LOOP` and `+LOOP` work with signed values. It's indisputable that working with signed values arises some problems, but defining this an "*horror*" seems quite exaggerated. Anyway, his anger is directed to the then-to-come Forth-83 `LEAVE` version which is, in his own words, "*intolerable*".

Probably there is some humour in this article that I, as an Italian, cannot fully understand. In any case, my personal point of view is that the `LEAVE` in Forth-79 may be bettered, yes, but it's not an horror.

2.5. Some `VOCABULARY` problems

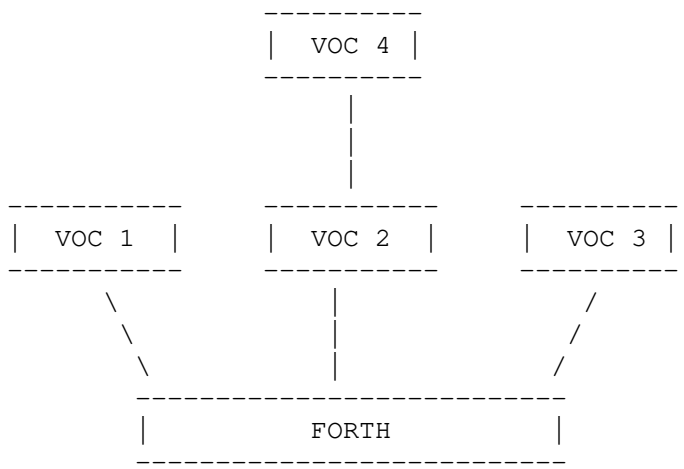
2.5.1. Chaining

The pure definition contained in the Standard: "*In lieu of any further specification, new vocabularies 'chain' to FORTH. That is, when a dictionary search through a vocabulary is exhausted, FORTH will be searched*" makes the `VOCABULARY` definition quite incomplete, someway a little weird, and leaves space to speculation.

This definition seems confirmed in the Forth-83 manual, where William F. Ragsdale writes about Forth-83: "*The method of selecting the order in which the dictionary is searched has grown from unchained vocabularies to the present* [Forth-83] *use of chained vocabularies*" (forgetting though that fig-Forth was a system that used chained vocabularies).

The only conclusion here is that in the Forth-79 Standard didn't require vocabularies to be explicitly chained, but they must be in any implementation, as I will demonstrate.

Let's build a model of vocabularies. The design model example is the following:



`VOC1`, `VOC2` and `VOC3` are bound to `FORTH` and `VOC4` is bound to `VOC2` (that is created into it).

Following strictly the Forth-79 definition, every search is limited to the `CONTEXT` and `FORTH` dictionaries, so when the user is in `VOC1`, `VOC2` or `VOC3` the definition guarantees that the `CURRENT` and `FORTH` dictionaries are found; but the same cannot be said for `VOC4`: when the user is in `VOC4`, the words in `VOC2` are not clearly guaranteed to be found (`VOC2` is not `CURRENT` and is not `FORTH`). The definition is therefore not exhaustive; and, what does "*in lieu of any further specifications*" mean? Isn't this a Standard?

Shouldn't it have all the needed specifications?

The only sage proposal is that vocabularies are chained according to their creation, and the engine traverses dictionaries in turn (the chain is not CURRENT 'and' FORTH, but the whole path going from CURRENT, through all the intermediate dictionaries, to FORTH).

In qartus vocabularies are built so that this unclear rule is removed: when in VOC4, this is searched first, then VOC2, and finally FORTH, which remains always in the background. Words in VOC1 or VOC3 are not available, of course.

Technically speaking, every word (see the Appendix) has a hardwired vocabulary reference coded into it. This is the reference that makes a word available or not. If the chain is VOC4-VOC2-FORTH (actually coded internally with something like 5-3-0), words in unreferenced vocabularies in the chain are not seen (for instance, EDITOR, which has reference 1, is not available during search).

The fact the Standard definition is incomplete does not imply that Forth-79 implementers could ignore such facts. I didn't.

2.5.2. Immediacy

In qartus, vocabularies created with VOCABULARY are not IMMEDIATE by default. Actually, this option was not specified by the Standard, and different examples in the Forth literature use IMMEDIATE after each VOCABULARY, suggesting that this feature was not a requirement but it was anyway necessary.
Anyway, in the Forth-83 Standard, chapter 5, it's written: "*Previously standardized words continue in their use:* VOCABULARY*,* FORTH*,* DEFINITIONS*, and* FORGET*. However, this proposal* [the Forth-83] *assumes that vocabulary names are not* IMMEDIATE". This statement implies that in (some) Forth-79 compilers, vocabularies were IMMEDIATE, though there is no evidence of this in the Forth-79 Standard document.

The solution, for me, is to leave the programmer the greatest freedom; if a VOCABULARY has to be immediate, it is declared as such (using IMMEDIATE), but if this is not required by its design, IMMEDIATE is not necessary, and no error condition exists.

2.6. A word about LITERAL (and DLITERAL)

The specifics for LITERAL in the Standard didn't set it as a compile-only word,; they require that LITERAL "*If compiling, then compile the stack value* n *as a 16-bit literal, which when later executed, will leave* n *on the stack*". This definition does not include the behaviour during execution: should n be maintained? Or is it consumed? Or should an error condition be issued? The solution I found was the one later applied in Forth-83, and probably left behind in the Forth-79 Standard: setting LITERAL as compile-only. Simple. Elegant.

Of course, the same concept was applied to 2LITERAL.

3. Words Reference

3.1. Notation in the Reference Manual

Each word is structured according to the following rules:

- The word name is in **CAPITAL** in ASCII alphabetical order. qartus is anyway case insensitive.
- When the *stack notation* is reported, it shows the stack situation (before -- after); if an R: is reported, it refers to the return stack (R: before -- after); if CF: are reported, it refers to the control flow stack (CF: before -- after). The top of the stacks is always to the right.
- Occasionally, when relevant, the *word pronunciation name* is written in double quotes.
- The items in the stack notation may be one of the following *types*:[27]

```
c         Any 0..127 character (unsigned 7-bit integer)
b         Any 8-bit byte (signed or unsigned 8-bit integer)
n         Any signed 16-bit integer (one cell, two bytes)
d         Any signed 32-bit double integer (two cells, four bytes)
addr      Any memory reference as an unsigned 16-bit integer
index     A postfix numeric index may be applied (n1,ud2,..)
u         Prefixes any unsigned number (ub,un,ud)
f         A boolean flag (zero, non-zero)
<name>    A dictionary word name
<file>    A file name (following own OS rules - not explained here)
<text>    Text following word in the buffered input stream
cccc      Any arbitrary series of ASCII characters (string)
file-id   An unsigned integer as a file channel identifier
fam       A constant as file access method (1=Read, 2=Write)
```

- Some *attributes* may be applied to the word:

```
C         The word may only be used during
          compilation of a colon definition.
I         Indicates that the word is immediate
          and will execute during compilation,
          unless special action is taken.
E         Intended for execution only,
          unless special action is taken.
S         The word is super-immediate.
U         A user variable (range is always specified).
```

- Some *source specifier* is applied to words:

```
COM       The word is a common usage word.
D79       The word belongs to the Standard Double Number Word Set.
F79       The word belongs to the Forth-79 Standard.
F83       The word belongs to the Forth-83 Standard.
F94       The word belongs to the ANSI 1994 Standard.
FIG       The word is peculiar to fig-Forth.
RWS       The word belongs to the 79-Standard Reference Word Set
QFT       The word is peculiar to qartus forth.
```

See also the qartus man page, which lists all the built-in words.

[27] In the rest of this chapter, the term 'cell' will always refer to a couple of 8-bit bytes (that is to 16 bits). Thanks to Ian Jones for pointing out this subject to me during the development phase.

3.2. The Dictionary

Here follow all the definitions of the words in the built-in dictionary, in alphabetical order.

! `n addr --` `"store"` F79

Store `n` at `addr`.

!CSP `--` `"store-c-s-p"` FIG

Save the stack position in `CSP`. Later on, this datum may be checked with `?CSP`, and an error is issued in case it differs from the current stack position. It can be used as a compiler security check.

`ud1 -- ud2` `"sharp"` F79

Generate from an unsigned double number `ud1`, the next ASCII character which is placed in an output string. Result `ud2` is the quotient after division by `BASE` and is maintained for further processing. Used only between `<#` and `#>`.

#> `ud -- addr n` `"sharp-greater"` F79

End numeric output conversion. Drop `ud` and leave the text address and character count of the string, suitable for `TYPE`. The character count is also saved in `FLD`. Used only to end a sequence begun with `<#`.

#BUFFERS `-- addr` U `"sharp-buffers"` FIG

Leave the address of a variable containing the number of active screen buffers in the system, in the range {0..3}. The user should not change this variable, because it is updated by the system.

#CLOCK `-- addr` U `"sharp-clock"` QFT

Leave the address of a variable containing the state of the `CLOCK` output; in case it is null, `CLOCK` will return a value to be read as milliseconds, since qartus's start; in case it is not null, the value returned by `CLOCK` is to be read as microseconds (1 microsecond = 1 millionth second).

#S `ud -- 0 0` `"sharp-s"` F79

Convert all digits of an unsigned 32-bit number `ud`, adding each to the pictured numeric output text, until remainder is zero, which is left on the stack. A single zero is added to the output string if the number was initially zero. Used only between `<#` and `#>`.

#TIB `-- addr` U `"number-tib"` F83

Leave the address of a variable containing the number of cells occupied in the text input buffer. The range of `#TIB` is {1..1024}.

#VOCS `-- addr` U `"sharp-vocs"` FIG

Leave the address of a variable containing the number of created vocabularies in the range {2..31}. The count always includes the `FORTH` and `EDITOR` vocabularies, so its lowest value is always 2. The user should not change the value of this variable, because it is updated by the system.

' `-- addr` `I "tick"` `F79`

Used in the form:

```
' <name>
```

If executing, leave the Parameter Field Address of the next word accepted from the input stream. If compiling, compile this address as a literal; later execution will place this value on the stack. An error condition exists if not found after a search in the CONTEXT and FORTH vocabularies. Within a colon-definition `' <name>` is identical to `[ ' <name> ] LITERAL`.[28]

(`--` `I "paren"` `F79`

Used in the form:

```
( <text>)
```

Accept and ignore characters from the input stream, until one next right parenthesis is found followed by a whitespace or the end-of-line[29]. A blank after the leading parenthesis is required. If the input stream is exhausted before the right parenthesis, the comment mode remains active until a manual right parenthesis is typed.

(+LOOP) `n1 --` `C` `F79`
`R: n2 n3 -- n3 n2+n1 |`
`R: n2 n3 --`

The run-time procedure compiled by +LOOP: increment the loop index by n1 and test for loop completion. See +LOOP.

(ABORT) `--` `FIG`

Executed by ERROR when WARNING is -1. This is a default run-time procedure that simply calls ABORT.[30] The programmer may write his own (ABORT) routine, which will be the one called by the error management system in case of existing error condition. See ABORT.

(AGAIN) `n1 n2 --` `C` `RWS`

The run-time procedure compiled by AGAIN: jump back unconditionally to its proper BEGIN. See AGAIN.

(BEGIN) `C` `F79`

The run-time procedure compiled by BEGIN. It marks the BEGIN position. See BEGIN.

---

[28] As said earlier in this manual, the Parameter Field Address and the Code Field Address varies depending on the item type (words, variables, CREATEd words, colon words, etc..). See also FIND.

[29] The right parenthesis directly followed by a character is part of the comment, and does not close it. This lets using (and) inside the comment, in such a way:

```
( comment (by M.E. Myself); this is a comment too )
```

with no risks. This is a minor difference with the Standard, which does not require this. This is, nonetheless, a feature that A. Winfield presents in his "*The Complete Forth*", and that probably was a feature of R-Forth.

[30] It's been built as a safe measure against forgotten implementation of (ABORT) in case the alternative error management is chosen.

(BLOCK-READ) un1 un2 -- addr | 0 QFT

Executed by BLOCK: read the block file un1 from disc and copy it to screen buffer un2. Leave the screen buffer address addr when done. The range of un2 is {1..3}, otherwise no action is taken and zero is left. Watch out: the previous content of the screen buffer is lost, because this procedure doesn't control if the destination screen buffer is updated or not, a type of control usually performed by BLOCK; SCR is not updated.[31] BFR is set to un2. See BLOCK.

(BLOCK-WRITE) un1 un2 -- f QFT

Executed by BLOCK and SAVE_BUFFERS: write the content of screen buffer un2 to disc block file un1. Leave true in case of success. The range of un2 is {1..3}, otherwise no action is taken and zero is left as flag. BLK, SCR and BFR are not changed. See BLOCK.

(CASE) C F94

The run-time procedure compiled by CASE. It marks the position of the current CASE clause. See CASE.

(COMPILE) C F79

The run-time procedure compiled by COMPILE: physically compiles the word address. See COMPILE.

(DO) n1 n2 -- C F79
R: -- n1 n2

The run-time procedure compiled by DO: move the loop control parameters to the return stack. See DO.

(DOES>) C F79

The run-time procedure compiled by DOES>. The Code execution token of this word is compiled just before the DOES> part of the defining word, signaling both the start of the following DOES> section (that will be copied to the defined word when the defining word is executed) and also the end of the executable part of the defining word. See DOES>.

(ELSE) f -- f C F79

The run-time procedure compiled by ELSE: test a copy of the flag f and if true jumps to THEN, otherwise continues execution. See ELSE.

(ENDCASE) n -- C F94

The run-time procedure compiled by ENDCASE: drop the comparing value n. See ENDCASE.

(ENDOF) CF: f -- C F94

The run-time procedure compiled by ENDOF: test the flag f in the control flow stack and if true jump to ENDCASE, otherwise continue execution. See ENDOF.

[31] I must warn you that, since SCR is not updated by the usage of (BLOCK-READ), an improper use of this word may cause overwriting a buffer that you may not have saved yet.

(IF) `n --` C F79
`CF: -- f`

The run-time procedure compiled by `IF`: test the value `n`; if true, continue execution leaving a true flag on the control flow stack; if false, jump to `ELSE` pushing a false value on the control flow stack. See `IF`.

(LOOP) `--` C F79
`R: n1 n2 -- n2 n1+1`
`R: n1 n2 --`

The run-time procedure compiled by `LOOP`: increment the loop index and test for loop completion. See `LOOP`.

(OF) `n1 n2 -- n1` C F94
`CF: -- f`

The run-time procedure compiled by `OF`: test `n2` against a copy of `n1`; if they match, continue execution, otherwise jump to the relative `ENDOF`. See `OF`.

(REPEAT) C F79

The run-time procedure compiled by `REPEAT`: jump back unconditionally to its proper `BEGIN`. See `REPEAT`.

(S `--` I "paren" QFT

This is a variant of `(` that skips text until the first closing and unique `)`, treating the inner text as a comment. The difference is not operational, but it's useful for sources readers, to distingues a stack comment from any other comment, and it may have a practical usage if you want to read files and parse the stack comments differently.

(SLITERAL) `-- addr n` I,C QFT

A primitive procedure that looks ahead in the next compiled memory region to find a compiled string, leaving on the stack the address and count of that string, suitable e.g. for `TYPE`.[32]

(SOUND) `n1 n2 --` I,C QFT

A primitive to produce a sinusoidal sound with a given frequency and a given duration. Parameters are interpreted this way: `n1` is the duration in milliseconds, while `n2` is the frequency in Hertz.

For instance, to produce the A note, useful to tune a guitar, use the following command:

```
2000 440 (SOUND)
```

This sequence will produce a 2 seconds A tone. Remember that, to use this, the

This word, quite experimental and not portable, was added in the form of a primitive to let the programmer create its own music or bell words (and use for instance a basic `SOUND` word).

[32] This word has one immediate practical usage: in `qlibrary.ft` the `SLITERAL` word is built around this primitive. About this matter read the chapter "Making Sound with qartus".

(THEN) CF: f -- C F79

The run-time procedure compiled by THEN: drop the flag on the control flow stack and continue execution. See THEN.

(UNTIL) f -- C F79

The run-time procedure compiled by UNTIL: test f, and if true continue execution, otherwise jump back to its proper BEGIN. See UNTIL.

(VOCABULARY) C F79

The run-time procedure compiled by VOCABULARY: set CONTEXT to the compiled vocabulary code. See VOCABULARY.

(WHILE) f -- C F79

The run-time procedure compiled by WHILE: test f, and if true continue execution, otherwise jump forward to its proper REPEAT. See WHILE.

(WORD) c -- addr C F79

The run-time procedure compiled by WORD: set c as the delimiter character; parse the input stream until the delimiter character is found (or until the end of stream if c=1) and copy the characters starting at addr+1; set addr to the string count; return addr, the counted string. See WORD.

***** n1 n2 -- n3 "times" F79

Leave the arithmetic signed product n3 of n1 times n2.

***/** n1 n2 n3 -- n4 "times-divide" F79

Multiply n1 by n2, divide the result by n3 and leave the quotient n4. n4 is rounded toward zero. The product of n1 times n2 is maintained as an intermediate 32-bit value for greater precision than the otherwise equivalent sequence: n1 n2 * n3 /.

***/MOD** n1 n2 n3 -- n4 n5 "times-divide-mod" F79

Multiply n1 by n2, divide the result by n3 and leave the remainder n4 and quotient n5. A 32-bit intermediate product is maintained as for */MOD for greater precision than the otherwise equivalent sequence: n1 n2 * n3 /MOD. The remainder has the same sign as n1 * n2.[33]

+ n1 n2 -- n3 "plus" F79

Leave the arithmetic sum n3 of n1 plus n2. Sum is always modulo 65536 using signed values.

+! n addr -- "plus-store" F79

Add n to the 16-bit value at the address addr, by the convention given for + .

[33] The standard reports here: "*The remainder has the same sign as n1*". It's clearly a typo, since the first operation is a multiplication, and if n2 is negative, it changes the sign of n1 (-n1*-n2=n1*n2, n1*-n2 = -n1*n2), so that in these cases, the remainder must be in accordance with n1*n2 rather than with n1 alone, and this in accordance with what is written for /MOD. Thanks to Ian Jones, who pointed out this problem.

+LOOP n -- I,C "plus-loop" F79

Add the signed increment n to the loop index using the convention for +, and compare the total to the limit. Return execution to the corresponding DO until the new index is equal to or greater than the limit (n>0), or until the new index is less than the limit (n<0). In case the increment n is zero, the cycle executes once only. Upon the exiting from the loop, discard the loop control parameters, continuing execution ahead. Index and limit are signed integers in the range {-32768..32767}.[34].

, n -- "comma" F79

Store n into the next available dictionary memory cell, advancing the dictionary pointer.

- n1 n2 -- n3 "minus" F79

Subtract n2 from n1 and leave the difference n3. Difference is always modulo 65536 using signed values.

--> -- I "next-block" RWS

Continue interpretation on the next sequential block (e.g. if --> is found on block file 100, block file 101 is loaded). No error condition exists if no more block files are present in qartus's directory, or if the next block file does not exist: the compilation will simply stop: this is a difference with the Forth-79 Standard. It must NOT be used within a colon definition that crosses a block boundary: this is another difference with the Forth-79 Standard.[35]

-TRAILING addr n1 -- addr n2 "dash-trailing" F79

Adjust the character count n1 of a text string beginning at addr, to exclude from the output the final trailing blanks. If n1 is zero or the entire string consists of spaces, n2 is zero. An error condition exists if n1 is negative.

. n -- "dot" F79

Display n converted according to BASE in a free field format with one trailing blank. Display a negative sign when n is negative.

---

[34] The Standard reports here for +LOOP: "*It is a historical precedent that the limit for* n<*0 is irregular. Further consideration of the characteristic is likely.*"). Well, the article by Robert L. Smith titled "*FORTH Standard Corner - DO, LOOP, and +LOOP*", on *Forth Dimensions* vol. III N. 6, presents a number of problems in the LOOP/+LOOP words as stated in the Standard, mainly the missed 0 increment case in +LOOP and the use of memory addressing with loop indices, which in the Standard are signed. See also the chapter "DO, LOOP and +LOOP problems".

[35] Far from being a programming misconception, this last difference is instead a moral rule that Leo Brodie exposes in "Thinking Forth" (ch. 5, page 143): "*...Some people consider the arrow to be useful for letting definitions cross screen boundaries. In fact --> is the only way to compile a high-level (colon) definition that occupies more than one screen, because --> is "immediate". But it's NEVER good style to let a colon definition cross screen boundaries. (They should never be that long!)*".

." -- I "dot-quote" F79

Interpreted or used in a colon definition in the form:

```
." cccc"
```

Accept the following text from the input stream, terminated by " (double-quote) or until the end of the input stream. If executing, transmit this text to the terminal. If compiling, compile so that later execution will transmit the text to the terminal. Up to 255 characters are allowed in the text. If the input stream is exhausted before the terminating double-quote, the string is terminated with a null character an no error condition exists (this is an enhancement with respect to the Forth-79).

.R n1 n2 -- "dot-r" RWS

Print n1 right aligned in a field of n2 characters, according to BASE. If n2 is less than n1 length in characters, no leading blanks are supplied.

.S -- "dot-s" COM

Display the data stack contents of 16-bit numbers, in a non-destructive way, according to BASE. The total number of cells (enclosed in < >) is printed, before the list, always in DECIMAL.

/ n1 n2 -- n3 "divide" F79

Divide n1 by n2 and leave the quotient n3. n3 is rounded toward zero. Division is symmetric.[36]

/MOD n1 n2 -- n3 n4 "divide-mod" F79

Divide n1 by n2 and leave the remainder n3 and quotient n4. n3 has the same sign as n1.

0< n -- f "zero-less" F79

True if n is less than zero (negative).

0= n -- f "zero-equals" F79

True if n is zero.

0> n -- f "zero-greater" F79

True if n is greater than zero.

1+ n -- n+1 "one-plus" F79

Increment n by one, according to the operation of +.

1- n -- n-1 "one-minus" F79

Decrement n by one, according to the operation of −.

[36] NOTE: the Forth-83 Standard changed the rounding behavior of the integer division, which was established to be floored, in which the remainder carries the sign of the divisor or is zero, and the quotient is rounded to its arithmetic floor. The ANSI 1994 decided that the symmetric division of Forth-79 was not bad, after all, and re-introduced it under the SM/REM word.

2! `d addr --` `"two-store"` D79

Store `d` in two consecutive cells beginning at `addr`.

2+ `n -- n+2` `"two-plus"` F79

Increment `n` by two, according to the operation of +.

2- `n -- n-2` `"two-minus"` F79

Decrement `n` by two, according to the operation of − .

2@ `addr -- d` `"two-fetch"` D79

Leave on the stack the contents of the two consecutive cells beginning at `addr`, as a double integer.

2CONSTANT `d --` `"two-constant"` D79

A defining word used in the form:

```
d 2CONSTANT <name>
```

to create a dictionary entry for `<name>`, leaving `d` in two consecutive cells of its parameter field. When `<name>` is later executed, `d` will be left on the stack.

2DROP `d --` `"two-drop"` D79

Drop the top double number on the stack.

2DUP `d -- d d` `"two-dupe"` D79

Duplicate the top double number on the stack.

2OVER `d1 d2 -- d1 d2 d1` `"two-over"` D79

Leave a copy of the second double number on the stack.

2ROT `d1 d2 d3 -- d2 d3 d1` `"two-rote"` D79

Rotate the third double number to the top of the stack.

2SWAP `d1 d2 -- d2 d1` `"two-swap"` D79

Exchange the top two double numbers on the stack.

2VARIABLE "two-variable" D79

A defining word used in the form:

```
2VARIABLE <name>
```

to create a dictionary entry of <name> and assign two consecutive cells for storage in the parameter field, setting the variable contents to zero. When <name> is later executed, it will leave the address of the first cell of its parameter field on the stack.

32! d -- "32-store" QFT

Store the double number on top of stack on a single memory unit (remember that one byte is seen as 8-bit but the system knows it as a 32-bit unit). Used to adjust the dictionary in relation to the special markers and link addresses present in a qartus compiled entry. This word is a qartus special word serving as a system helper.

32, d -- "32-comma" QFT

Allot the double number on top of stack in one unit in the dictionary, storing it as a 32-bit number (remember that one byte is seen as 8-bit but the system knows it as a 32-bit unit). Used to compile special markers and link addresses. This word is a qartus special word serving as a system helper.

32@ addr -- d "32-fetch" QFT

Leave on the stack as a double number the number contained in the unit addr (remember that one byte is seen as 8-bit but the system knows it as a 32-bit unit). Used to parse the dictionary entries and look for the special markers and link addresses present in a qartus compiled entry. This word is a qartus special word serving as a system helper

79-STANDARD -- F79

Assure that the environment executing the program is a Forth-79 Standard system, otherwise an error condition exists (it's supposed to be missing in a non-compliant system). Also, reset the fig-Forth compatibility flag FIG-STATE to zero, assuring that a normal Forth-79 behaviour is performed from now on.

: -- E "colon" F79

A defining word executed in the form:

```
: <name> ... ;
```

Select the CONTEXT vocabulary to be identical to CURRENT. Create a dictionary entry for <name> in CURRENT, and set compile mode. Words thus defined are called 'colon-definitions'. The Link-Field-Addresses of subsequent words from the input stream which are not immediate words are stored in the dictionary to be executed when <name> is later executed. IMMEDIATE words are executed as encountered.

If a word is not found after a search of the CONTEXT and FORTH vocabularies, conversion and compilation of a literal number is attempted, with regard to the current BASE; that failing, an error condition exists, an error message is printed and interpret mode is set, removing all effects established by current compilation.

; -- I,C "semi-colon" F79

Terminate a colon definition and stop compilation. If compiling from mass storage and the input stream is exhausted before encountering **;** an error condition exists.

;S -- E "semi-s" RWS

Stop the interpretation of a block. In qartus, the function of this word has been enhanced to allow also stopping the interpretation of a file.

If used into a compilation word, acts as a word ender (compiles the word ender token and a final 0); the code after **;**S is compiled and occupies its memory, but it's never executed, nor shown by SEE. It can thus be used to shorten the execution of a word under design, avoiding the unnecessary code for the current investigation.

< n1 n2 -- f "less-than" F79

True if n1 is less than n2.
-32768 32767 < returns true.
-32768 0 < returns true equally, and thus the two numbers are distinguished (as required by the Standard).

<# -- "less-sharp" F79

Initialize pictured numeric output. The words:

```
<# # #S HOLD SIGN #>
```

are variously used to specify the conversion of a double-precision number into an ASCII character string stored in right-to-left order.

= n1 n2 -- f "equals" F79

True if n1 is equal to n2.

> n1 n2 -- f "greater-than" F79

True if n1 is greater than n2.

>IN -- addr U "to-in" F79

Leave the address of a variable which contains the current character offset within the input buffer in the range {0..1023} bytes. See: WORD, (, **."**, FIND and #TIB.

>R n -- C "to-r" F79

Transfer n to the return stack. Every >R must be balanced by a R> in the same control structure nesting level of a colon-definition.

? addr -- "question-mark" F79

Display the number at address, using the format of **.** (the dot).

?CSP -- "query-c-s-p" FIG

Issue an error message, followed by (ABORT), if stack position differs from value saved in CSP. It can be used during the compilation phase of a program, to check if stack value in compilation changes (and shouldn't). In conjunction with WARNING and (ABORT), ERROR can issue a proper and personal error message and/or run a proper and personal user procedure.

?DUP n -- n n "query-dupe" F79
0 -- 0

Duplicate n if it is non-zero.

?ERROR n f -- "query-error" FIG

In case flag f is true, leave n on the stack and call ERROR; otherwise drop both f and n and continue execution. See also the chapter "Error management".

?TERMINAL -- n | 0 "question-terminal" COM

Check if a key was pressed from keyboard, and return zero if not; otherwise return the character code of the key that was pressed (in the assumption that the NULL character cannot be obtained from a common keyboard, this value is always different from zero). This word is non-blocking.

@ addr -- n "fetch" F79

Leave on the stack the number contained in the cell starting at addr.

ABORT -- F79

Clear the data, return and control flow stacks, setting execution mode. Print a proper message. Return control to the terminal without printing the 'ok' string.

ABS n -- |n| "absolute" F79

Leave the absolute value of the number n.

AGAIN I,C RWS

Jump back unconditionally to the start of its proper BEGIN..AGAIN loop.

At compile time, AGAIN compiles the run-time word (AGAIN) and the jump address left on the control flow stack by BEGIN.

ALLOT n -- F79

Reserve n bytes in the dictionary. If it follows a recently defined word (with CREATE, for instance), this means setting its parameter field n bytes wide.[37]

AND n1 n2 -- n3 F79

Leave the bitwise logical 'and' of n1 and n2 as n3.

[37] This means of course that in the n bytes reserved by ALLOT can be stored at most n 8-bit numbers or characters, n/2 single numbers and n/4 double numbers.

ASCII -- c (executing) I RWS
-- (compiling)

Leave (as a 16-bit number) the 7-bit ASCII character value of the next non-blank character in the input stream. If compiling, compile it as a literal, which will be later left when executed.

B/BUF -- 1024 "buffer-bytes" RWS

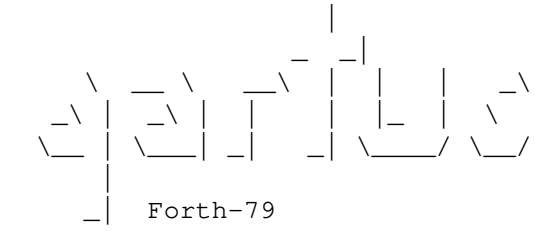
A constant leaving 1024, the number of bytes per block buffer, that is the byte count read from disc by BLOCK.

B/SCR -- 1 "screen-blocks" FIG

A constant leaving the number of blocks per editing screen. By convention, an editing screen is 1O24 bytes organized as 16 lines of 64 characters each. Each screen can hold one entire block.

BANNER -- QFT

Print the following banner:



to mark your program a qartus product!
In case of fig-Forth emulation, a proper string is appended (see FIG-STATE).

BASE -- addr U F79

Leave the address of a variable containing the current input-output numeric conversion base in the range {2..36}.[38]

BEEP -- QFT

Play a beep. This is a synonym of BELL (added for compatibility reasons). See BELL for all the technical stuff.

[38] The Forth-79 Standard specifies that BASE should have the range {2..70}; in modern computer science, bases greater than 36 (which is the sum of 10 ASCII digits plus 26 ASCII letters, either capital or lower) are meaningless, and practically not implementable with the ASCII set 33-127 (set of printable chars), unless you want to distinguish, for instance, 'a' from 'A', which is the best way to mess up things; besides, nowhere in the Standard is written which is the ordering of these 70 characters (ASCII? EBCDIC? Other?); creating one by purpose would cause a real messy and unportable criterion for number recognition. This is another (minor, I hope) difference with the Forth-79 Standard, confirmed also by most recent Forth Standards (e.g. ANSI 1994) that set BASE to be in the range {2..36}.

BEGIN I,C F79

Used in a colon-definition in the form:

```
BEGIN ... AGAIN or
BEGIN ... f UNTIL or
BEGIN ... f END or
BEGIN ... f WHILE ... REPEAT
```

Mark the start of a word sequence for repetitive execution. A BEGIN..AGAIN loop is repeated indefinitely, and it may be interrupted by EXIT or by a keyboard interrupt. A BEGIN..UNTIL loop (that may be spelled also as BEGIN..END, from an older tradition) will be repeated until flag f is true. A BEGIN..WHILE..REPEAT loop will be repeated until flag f is false. The words after AGAIN are never executed. The words after UNTIL or REPEAT will be executed when either loop is finished. The flags of WHILE and UNTIL are always dropped after being tested.

At compile time, BEGIN compiles the run-time word (BEGIN), and leaves its return address on the control flow stack.

BELL -- QFT

Play a beep[39]. The default beep is a sound of 880 Hz for about 140 milliseconds. See also BEEP that performs an identical task.

BFR -- U QFT

Leave the address of a variable containing the number of the current edited screen buffer. Every screen buffer/block word (BLOCK, BUFFER, LIST, EDIT) when invoked, set BFR to the next available screen buffer if the block was loaded, or to the correspondent screen buffer in case the block is already in memory. The range is {1..3}. The user can manually change BFR to edit another screen.

BL -- 32 "b-l" RWS

A constant that leaves the ASCII character value for blank (decimal 32).

BLK -- addr U "b-l-k" F79

Leave the address of a variable containing the number of the mass storage block being interpreted as the input stream. If the content is zero, the input stream is taken from the terminal. Range is {0..65535} as an unsigned single number (see BLOCK).

BLOCK un -- addr F79

Leave the address of the first cell in screen buffer for the block un. If the block is not already in memory, it is transferred from mass storage into whichever screen buffer has been least recently accessed. If the block occupying that screen buffer was UPDATEd (i.e. modified), it is rewritten onto mass storage before the new block-file un is read into the screen buffer. If mass memory reading is not possible (not-existing block or denied directory access), the screen buffer is filled with blanks[40]. If correct mass storage writing is not possible, an error condition exists. Only data within the latest screen buffer referenced by BLOCK is valid by cell address, due to sharing of the screen buffers. At all effects, BLOCK sets BFR, and un is associated to the screen addressed by BFR.[41]

---

[39] About this matter read the chapter "Making Sound with qartus".

[40] This is an important difference with the Standard, though applicable only in case of not-existing blocks, and not applicable in normal processing.

BUFFER un -- addr F79

Obtain the next screen buffer, assigning it to block un. The block is not read from mass storage. If the previous contents of the screen buffer has been marked as UPDATEd, it is written to mass storage to the correspondent block number. If correct writing to mass storage is not possible, an error condition exists. The address left is the first cell within the screen buffer for data storage. BUFFER sets BFR, and un is associated to the screen addressed by BFR.

BYE -- S COM

Immediately terminate current qartus session.[42] This word is a super-immediate word. If, in a list of files, BYE should appear in one but the last file, it is ignored. BYE works even in blocks, terminating the Forth session within the block interpretation.

The word BYE has a counterpart in the dictionary, to make it compilable with [COMPILE]; this is the only service word that has this feature. It was introduced to enable the conditional program ending, for instance after a proper user choice, to exit not only from the program, but from qartus too.

C! n addr -- "c-store" F79

Store the least significant 8-bits of n in the byte at addr.

C, n -- "c-comma" F79

Store the least significant 8-bits of n in the dictionary, advancing HERE by one.

C/L -- 64 "chars-per-line" COM

A constant containing the number of characters per line in a buffer screen.

C@ addr -- ub "c-fetch" F79

Leave on the stack the unsigned content ub of the byte at addr (with higher bits zero in a 16-bit field).

CASE -- F94

At compile-time, mark the start of a CASE..OF..ENDOF..ENDCASE structure. At run-time, continue execution.

CF-GET -- n QFT

Remove n from the control flow stack and put it to the data stack.

CF-PUT n -- QFT

Remove n from the data stack and put it to the control flow stack.

---

[41] **Important note**: all blocks are saved in a hidden directory, to prevent accidental deletion. This directory is $HOME/.qartus/blocks/ and is created by the option --config (see the man page for the Windows© configuration). In this directory, blocks are saved as pure textual files of 16 lines with 64 characters each, in total 1024 bytes (plus the line terminators) for each block. A very compact storage. You can copy, rename, move, delete, send, print and do any regular operation upon these files, since they are normal textual files. You can use any directory for saving blocks through the word USE.

[42] See also COLD.

CLOCK `-- ud` QFT

Return the quantity of milliseconds or of microseconds (1 millionth second) from the start of qartus. The returned value of `CLOCK` is a double number in the range {0..4294967295} to ensure the maximum range. The difference is governed by the variable `#CLOCK`: if it contains zero milliseconds are returned, otherwise microseconds are returned.

In the default milliseconds configuration (`#CLOCK` contains zero), this ensures a maximum runtime length of 4294967 seconds, which correspond to 71582 minutes, or 1193 hours, that is more than 49 days!

In the microseconds configuration (`#CLOCK` does not contain zero), this ensures a maximum runtime length of 4295 seconds, which correspond to about 71 minutes, or 1 hour and 11 minutes, but assures a greater precision with faster programs and very small intervals.

For an example of usage, take a look at the set of `TIMING`, `RESET-TIMER`, `TIMER` and `NTIMER` in the default library, with `CLOCK` as base words showing how to implement a benchmark tester. Change at will `#CLOCK` to contain 0 or 1, and see the difference (for instance the program `ybench.ft` in the files/ directory).

CLOSE-FILE `file-id -- 0` F94
`file-id -- 1`

Close the file identified by `file-id`. Return zero on success, one otherwise.

CMOVE `addr1 addr2 n --` `"c-move"` F79

Move the specified quantity `n` of bytes beginning at address `addr1` to memory at `addr2`. The contents of `addr1` is moved first, proceeding toward high memory. If `n` is zero or negative, nothing is moved.

COLD `--` S COM

This word is a super-immediate word used to restart immediately, rebuilding the default qartus's dictionary. This word does not properly belong to the dictionary.[43]

COMPILE C F79

When a word containing `COMPILE` executes, the cell value following the compilation address of `COMPILE` is copied (compiled) into the dictionary. i.e., `COMPILE DUP` will copy the compilation address of `DUP`.

`COMPILE [ 0 , ]` will copy zero.[44]

At compile time, `COMPILE` compiles the run-time word `(COMPILE)` and the compilation address (the Link Address or the built-in code) of the next word in the input stream.

[43] See also `BYE`. Remember that `COLD` rebuilds only the default built-in dictionary, and not the words you might have loaded through `LOAD`, `ENSURE`, `REQUIRE` or `INCLUDE` or the resource file `$HOME/.qartus.rc`.

[44] This is required and explicitly specified in the Standard, but I observe here that this definition is equivalent to `[ 0 , ]` (`COMPILE` omitted), because here `COMPILE` simply ignores what follows. The only practical note that can be derived from this is that `COMPILE` cannot compile `[`!

COMPILE-ONLY E COM

Mark the most recently made dictionary entry as a word which will be usable only into definitions and not directly from console.[45]

CONSTANT n -- F79

A defining word used in the form:

```
n CONSTANT <name>
```

to create a dictionary entry for <name>, leaving n in its parameter field. When <name> is later executed, n will be left on the stack.

CONTEXT -- addr U F79

Leave the address of a variable specifying the vocabulary index in which dictionary searches are to be made, during interpretation of the input stream; the index is in the range {0..30}.

CONVERT d1 addr1 -- d2 addr2 F79

Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. addr2 is the address of the first non-convertible character. This word accepts negative numbers, that are added to d1 according to the usual laws of arithmetic.[46]

COUNT addr -- addr+1 ub F79

Leave the address addr+1 and the character count of text beginning at addr. The first byte at addr must contain the unsigned character count ub, but if ub is zero or greater than 255, a condition of error exists.

CR -- "c-r" F79

Transmit a carriage-return and line-feed to the terminal.

CREATE -- F79

A defining word used in the form:

```
CREATE <name>
```

to create a dictionary entry for <name>, without allocating any parameter field memory. When <name> is subsequently executed, the address of the first cell of <name>'s parameter field is left on the stack.[47]

---

[45] See the related word BUFFERED in the library qlibrary.ft, and study how it is implemented. COMPILE-ONLY could be written in Forth in a similar way with 8 (the compilation bit) as the bit selector.

[46] I note that nowhere in the Standard is reported if dots in the string are accepted or not. Anyway, I enabled this feature in qartus because it seems logical that, if a number from keyboard contains dots and is accepted as a double number, the same must be said if the source of the number is a string in memory (it may have been typed by the user). Moreover, the Standard does not specify if the leading minus (-) is accepted (even if it seems logical it does). In any case, in qartus, a leading minus or even a leading plus (as for the common keyboard input) is accepted and works as expected (producing a negative number in case of minus and ignored in case of plus).

[47] The compile-time version of this word was known, before the Forth-79 Standard (the 1978 fig-Forth model, actually), as <BUILDS, and this explains the "strange" format of DOES>; the construction was in fact made with

CSP -- addr U FIG

Leave the address of a variable temporarily specifying the stack pointer position, for compilation error checking. See !CSP and ?CSP. The range of CSP is {0..65535}.

CURRENT -- addr U F79

Leave the address of a variable specifying the vocabulary index into which new word definitions are to be entered; the index is in the range {0..30}.

D* d1 d2 --- d3 "d-times" COM

Leave the signed product d3 of the two signed double numbers d1 and d2.

D+ d1 d2 --- d3 "d-plus" F79

Leave the arithmetic signed sum d3 of d1 plus d2.

D− d1 d2 -- d3 "d-minus" D79

Subtract d2 from d1 and leave the difference d3.

D. d -- "d-dot" D79

Print d converted according to BASE in a free field format, with one trailing blank at the end. Display the sign in case it's negative.

D.R d n -- "d-dot-r" D79

Print d converted according to BASE, right aligned in a field n characters wide. Display the sign in case it's negative. If n is negative or smaller than the characters required for d, no leading spaces are given.

D/ d1 d2 --- d3 "d-divide" COM

Divide d1 by d2 and leave the quotient d3. d3 is rounded toward zero.

D0= d -- f "d-zero-equals" D79

Leave true if d is zero.

D< d1 d2 -- f "d-less" D79

True if d1 is less than d2.

D= d1 d2 -- f "d-equal" D79

True if d1 equals d2.

... <BUILDS ... DOES> ...

and this makes a perfect sense. Now, <BUILDS and DOES> in fig-Forth bear some differences with the Forth-79 counterparts; in particular, <BUILDS allocates one cell in the Parameter Field Address of the newly created words, while CREATE does not allocate anything. Besides, DOES> in Forth-79 is IMMEDIATE, while in fig-Forth it's not.

DABS d -- |d| "d-absolute" D79

Leave the absolute value of a double number d.

DATE -- n1 n2 n3 QFT

Leave on stack current year n1, current month n2 and current day n3 on top. The year is in four digits (e.g. 2019), n2 is in the range {1..12}, n3 in the range {1..31}.

DEBUG 1|0 -- QFT

Enable/disable the debug mode. TRACE and DEBUG mutually exclude themselves, because they interfere with each other operations. So use one or the other, on the need.

To enable debugging, use constants ON and OFF like this:

```
ON DEBUG
```

to enable debugging, and

```
OFF DEBUG
```

to disable debugging. In either cases TRACE is disabled.

DEBUG-STATE -- U F79

Leave the address of a variable containing the current debug state, where 0 means debug off, and not-zero means debug on. This value may read or changed, and the change will influence the debug.

DECIMAL -- F79

Set the input-output numeric conversion base to 10.

DEFINITIONS -- F79

Set CURRENT to the CONTEXT vocabulary so that subsequent definitions will be created in the vocabulary previously selected as CONTEXT.

DEPTH -- n F79

Leave the number of the quantity of 16-bit cells contained in the data stack, before n is added (that is half of the number of occupied bytes).

DLITERAL d -- I,C FIG

If compiling, then compile the double stack value d as a 32-bit literal, which when later executed will leave d on the stack as a double number.[48]

[48] More recent versions of Forth use the syntax 2LITERAL, maybe for assonance with 2VARIABLE and 2CONSTANT. But I decided to use the fig-Forth version of the word. If you need to use 2LITERAL, please ENSURE it. It is contained in the qlibrary.ft file.

DMAX d1 d2 -- d3 "d-max" D79

Leave d3, the greater of the two double numbers d1 and d2.

DMIN d1 d2 -- d3 "d-min" D79

Leave d3, the lesser of the two double numbers d1 and d2.

DNEGATE d -- -d F79

Leave the two's complement of a double number d, i.e. the difference 0 less d.

DO n1 n2 -- I,C F79

Used in a colon-definition:

```
DO ... LOOP or
DO ... +LOOP
```

Begin a loop which will terminate based on control parameters. The loop index begins at n2, and terminates based on the limit n1 (being n1-1 the last evaluated number).[49] At LOOP or +LOOP, the index is modified by a positive or negative value. The range of a D..LOOP is determined by the terminating word.

DO..LOOP may be nested. There are no limits (except the stacks overflow) to the loop nesting. Index and limit are 16-bit signed integers.

At compile time, DO compiles the run-time word (DO) and leaves its address on the control flow stack.

DOES> I,C "does" F79

Define the run-time action of a word created by a high-level defining word. Used in the form:

```
: <name> ... CREATE ... DOES> ... ;
```

and then

```
<name> <namex>
```

Mark the termination of the defining part of the defining word <name> and begins the defining of the run-time action for words that will later be defined by <name>. On execution of <namex> the sequence of words between DOES> and ; are executed, with the address of <namex>'s parameter field on the stack.

[49] The index of the DO..LOOP cycle can be used to inspect the whole memory, both in signed and unsigned mode:
(signed mode) `32767 -32767 DO I 32767 + .. LOOP`
(unsigned mode) `65535 0 DO I .. LOOP`
In both cases, the memory range is {0..65534}, due to the peculiar structure of DO..LOOP, that does not evaluate last limit value. And being 65535 the higher unsigned number, the last memory slot must be inspected out of the DO..LOOP cycle.

DPL `-- addr` U `"f-l-d"` RWS

A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used to hold output column location of a decimal point, in user generated formatting. The default value on a single number input is -1. The range of `DPL` is {0..255} as an unsigned value.[50]

DROP `n --` F79

Drop the top number from the stack.

DU< `ud1 ud2 -- f` `"d-u-less"` D79

True if `ud1` is less than `ud2`. Both numbers are unsigned.

DUMP `addr un --` RWS

List the contents of `un` bytes starting from the address at `addr`, with addresses and values expressed in current `BASE`. Each line of values is preceded by the address of the first value and followed by a pictured image of the ASCII characters in the cells (a dot for each non-printable value). Each `BASE` defines its own number of bytes per row (from 1 in `BASE=2` to 8 in `BASE=36`). Here's an example of output in hexadecimal:

```
hex  ok
1 41 dump
0001 00000000 00000000 00000000 0AACAADA 00000004 00000062 00000061 .....ba
0008 00000073 00000065 00000064 00000010 00000000 00000000 00000001 sed....
000F 00000000 00000000 0AACAADA 00000005 00000073 00000074 00000061 ....sta
0016 00000074 00000065 00000064 00000000 00000000 00000000 0000000E ted....
001D 00000000 00000000 0AACAADA 00000003 0000003E 00000069 0000006E ....>in
0024 00000064 00000009 00000000 00000000 0000001C 00000000 00000000 d......
002B 0AACAADA 00000003 00000063 00000073 00000070 00000064 00000000 ..cspd.
0032 00000000 00000000 00000028 00000000 00000000 0AACAADA 00000004 ..(....
0039 00000073 00000070 00000061 0000006E 00000064 00000000 00000000 spand..
0040 00000000 00000034                                             .4
```

DUP `n -- n n` `"dupe"` F79

Leave a copy of the top stack number.

[50] The Reference Word Set reports the following definition for `DPL`: "*A variable containing the number of places after the fractional point for output conversion. If* `DPL` *contains zero, the last character output will be a decimal point. No point is output if* `DPL` *contains a negative value.* `DPL` *may be set explicitly, or by certain output words, but is unaffected by number input*".
Since no Standard word prints out the fractional point, there are two alternatives: either the implementation of this word changes also the various printing words (namely `D.R`, `D.` and `UD.`) - but this violates someway the Standard itself; or this is a variable usable by the programmer for printing the fractional point when desired, for *user words only*!
I set up `DPL` in qartus following this second alternative (which follows the fig-Forth and R-Forth - according to some examples in the Alan Winfield book, in chapters 8.1 and 8.5); that is, `DPL` is updated after each double number input, for checking the user input and acting consequently. This can be considered a minor difference with the Standard.

EDIT un -- E QFT

Start an editing session with the screen editor of block un (which corresponds to a block number), setting SCR to contain un. un is unsigned. EDIT uses BLOCK to load the block in memory, in the first available screen buffer, whose value is yield by BFR. At all effects, EDIT updates SCR and BFR.

The EDIT program is a standalone application, and it has no relation, nor common operational commands, with the built-in line editor contained in the EDITOR vocabulary.

See the chapter "The screen editor in details" ahead.

EDITOR I RWS

The name of the editor vocabulary. When this name is executed, EDITOR is established as the CONTEXT vocabulary and all the line editor commands become available.

ELSE -- I,C F79

Used in a colon-definition in the form:

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

ELSE executes after the true part following IF. The word ELSE forces execution to skip till just after THEN in case the flag left by *F* is true. It has no effect on the stack. (see IF).

At compile time, ELSE compiles the run-rime word (ELSE), placing its address on the reserved byte following (IF), and leaves its own address on the control flow stack.

EMIT b -- F79

Transmit the character b to the terminal (as 8-bit bytes with higher bits set to zero), according to the ASCII table.[51]

EMPTY-BUFFERS -- F79

Mark all block buffers as empty, without affecting their actual contents. UPDATEd blocks are not written to mass storage.

END f -- I,C RWS

A duplicate definition for UNTIL. See UNTIL.

ENDCASE addr -- I,C F94

At compile-time, resolve all branches in the case structure, such that the address beyond ENDCASE becomes the destination of of all branches after each ENDOF. At run-time, discard the original comparison value and continue execution.

[51] The Standard specifies for EMIT the characters range in the {0..127} region of the ASCII table. I built EMIT to use the full {0..255} range, but with the advice that the ASCII table in the range {1..127} is standard, and surely portable among various system, while the range {128..255} is system-dependent. This is a minor deviation from the Standard.

ENDIF -- I,C F94

A duplicate definition for THEN. See THEN.

ENDOF -- I,C F94

At compile-time, provide the location address following ENDOF as the destination address needed by the corresponding OF, and place a new forward reference to the next OF or default structure. At run-time, jump to the next OF or default structure.

ENSURE E QFT

Used in the form:

```
ENSURE <name>
```

Just like REQUIRE, parse file qlibrary.ft (looking for it in the default library file) then, if the word <name> is not present in the dictionary, read and compile it, along with all the dependencies words that are in the same library and that are not yet in the dictionary. ENSURE cannot be compiled, and is executable-only.

See the note in REQUIRE, which contents apply to ENSURE too.

ERNUM -- n QFT

A constant leaving the number of the default error messages hardwired in current qartus implementation. It may be used to know the cycle values needed to print or check all error messages (see MESSAGE and the word PERRORS in the library).

ERR -- addr U QFT

Leave the address of a variable containing the last error code retrieved by ERROR. The default value is -1. The range of this variable depends on the contest: if WARNING is 1, it is set to a value in the range {0..15}; if WARNING is 0, it is a positive value in the range of the default error strings array; if WARNING is -1, the range is {-32768..32767} and the user is free to use his own positive or negative codes to manage his error routines. This variable should not be changed by the user, because it is changed by ERROR and reset to -1 in case of range or buffer error in ERROR.[52]

ERROR n -- FIG

Execute the user error notification. Set ERR to the value n. WARNING is then examined. If = 1, the text of line n, relative to the most recently loaded screen buffer, is printed, with n in the range {0..15}. If WARNING = 0, the error condition n is passed to the system error management procedure, simulating a real error condition, with n in the range established in errors.h. If WARNING is -l, (ABORT) is executed, which executes in turn the system ABORT. The user may change this by creating his own (ABORT), using ERR to retrieve the user error code. In case n is not in the defined ranges, a real error condition exists! See also the chapter "Error management".

[52] This is the case when you type, for instance, 587 ERROR with WARNING = 0 or 1, or when you type 13 ERROR with WARNING = 1 and no buffer available; you can then check ERR: if it is -1, the ERROR call went wrong, otherwise ERR has the error code value.

EXEC-ONLY E QFT

Mark the most recently made dictionary entry as a word which will be usable only from console and not into definitions.[53] Such words, with the notably exceptions of IMMEDIATE, COMPILE-ONLY and EXEC-ONLY itself) may be compiled using [COMPILE], but their behaviour is not guaranteed to work on runtime in the same way it runs when directly invoked.

EXECUTE addr -- F79

Execute the dictionary entry whose Code Field Address is on the stack (that is the start of a compiled region)[54].

EXIT -- C F79

When compiled within a colon-definition, terminate execution of that definition, at that point, by immediately reaching the end of the word definition. May not be used inside a DO..LOOP because it would leave unneeded values on the Return Stack (in this case LEAVE should be used to exit the loop anticipately).

EXPECT addr n -- F79

Transfer characters from the keyboard, beginning at addr, upward, until a "Return" or the count of n has been received. Take no action for n less than or equal to zero. A condition of error exists if n is greater than 255. Nulls are added at the end of the text up to the length n. Each typed key is echoed; the backspace can be used to retype mistaken text. The actual count of the typed characters (not necessarily equal to n) is stored in the user variable SPAN.

FALSE -- 0 COM

A constant leaving the false value.

FENCE U COM

Leave the address of a user variable containing an address below which the user is not allowed to FORGET. In order to use FORGET on an entry below this point, it is necessary to alter the contents of FENCE. If FENCE is null, the limit for FORGET is fixed to the last built-in word. The range of FENCE, since it's an address, is {0..65535} (unsigned number).

FIG-STATE -- addr U QFT

Return the address of the fig-Forth compatibility flag; when this contains zero, qartus behaves in the default way, as a Forth-79 environment; when it contains a value different than zero, qartus behaves more like fig-Forth for the differences that do exist between the two (to execute, if possible, programs written for fig-Forth avoiding much rewriting).

79-STANDARD resets the value of the variable putting 0 in FIG-STATE.

See also the chapter "fig-Compatibility".

[53] See the related word BUFFERED in the library qlibrary.ft, and study how it is implemented. EXEC-ONLY could be rewritten in Forth in a similar way with 64 (the exec bit) as the bit selector.

[54] See the relative article in this manual devoted to EXECUTE.

FILL `addr n ub --` F79

Fill memory beginning at address `addr` with a sequence of `n` copies of `ub`. If the quantity `n` is less than or equal to zero, take no action.

FIND `-- addr | 0` F79

Used in the form

`FIND <name>`

Leave the Link Field Address (a.k.a. Compilation Address) of the next word `<name>`, which is accepted from the input stream. If that word cannot be found in the dictionary after a search of `CONTEXT` and `FORTH` dictionary leave zero.

FIRST `-- addr` FIG

A constant leaving the address of the first byte of screen buffer #1. This constant is calculated at start and may vary if the stack or the return stack are re-dimensioned though the relative console options.

FLD `-- addr` U `"f-l-d"` RWS

A variable containing the field length of the latest output conversion through the `<# ... #>` sequence. The range of `FLD` is {0..255} as an unsigned number.

FORGET E F79

Execute in the form:

`FORGET <name>`

Delete `<name>` from the dictionary (`<name>` must be in the `CURRENT` vocabulary) and all words added to the dictionary after `<name>`, regardless of their vocabulary. If the word `<name>` is not find in the `CURRENT` or `FORTH` dictionaries, no error condition is generated.[55]

FORTH I F79

The name of the primary vocabulary. Execution makes `FORTH` the `CONTEXT` vocabulary. New definitions become a part of the `FORTH` vocabulary until a different `CURRENT` vocabulary is established. User vocabularies conclude by ultimately 'chaining' to `FORTH`, so it should be considered that `FORTH` is 'contained' within each user's vocabularies chain.

H. `n --` COM

Print `n` in hexadecimal.

---

[55] This is a minor difference with the Standard, that explicitly prescribes that "*Failure to find <name> in CURRENT or FORTH is an error condition.*". I set it as such in order to use `FORGET` as a simple `MARKER` (ref. ANSI 1994); simply `FORGET` a specific marker word, and then redefine it at the very beginning of your program; this will let you reload unlimited times the same dictionary without redefinitions and without the necessity to manually `FORGET` specific sections of the dictionary. This is useful during the implementation phase, e.g.:

```
FORGET MARKER
: MARKER ;
```

Of course, in the design and implementation phase, `FENCE` must be avoided; it becomes necessary only when you are sure your program works (when you can also decide to remove the `MARKER` section).

HERE -- addr F79

Return the address of the next available dictionary location.[56]

HEX -- COM

Set BASE to decimal 16 (hexadecimal base). See H..

HLD -- addr U FIG

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

HOLD c -- F79

Insert char c into a pictured numeric output string. May only be used between <# and #>.

I -- n C F79

Copy the current loop index onto the data stack[57]. May only be used in the form:

```
DO ... I ... LOOP or
DO ... I ... +LOOP or
```

the word I always refers to the most inner loop in case of nested DO..LOOPS.

IF f -- I,C F79

Used in a colon-definition in the form:

```
f IF ... ELSE ... THEN or
f IF ... THEN
```

In the first form, if flag f is true, the words following IF are executed and the words following ELSE are skipped; if f is false, the words following IF are skipped and the words after ELSE are executed. The second form shows that the ELSE part is optional; in this case, if f is false, words between IF and THEN are skipped and never executed. IF..ELSE..THEN conditionals may be nested.

At compile-time, IF compiles the run-time word (IF) and reserves space for a 32-bit address, that will be later used by ELSE or THEN.

[56] As said earlier in this manual, since the program dictionary starts at location 1 of the memory, the quantity of occupied cells in the program memory can be calculated with HERE-1.

[57] In his book "*The complete Forth*", in paragraph 9.6. Alan Winfield presents his STEP..DOWN procedure, which is a sort of a one-way DO..LOOP. In this scheme he never uses DO..LOOP, but his compiler lets him use I inside a word and outside of a DO..LOOP. This contradicts the Standard Forth-79, which specifies that I (and J) must be used "*only in the form*" of a DO..LOOP. Anyway, the task of I outside of a DO..LOOP scheme can be safely performed by R@, and at all effect, I is equal to R@ for all licit usages. This is confirmed indirectly by the fig-Forth definition of I: "*Used within a* DO-LOOP *to copy the loop index to the stack.*". Other usages were left to implementors, but the Forth-79 later removed this faculty.

IMMEDIATE E F79

Mark the most recently made dictionary entry as a word which will be executed when encountered during compilation rather than being compiled.[58]

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 `[COMPILE]`.

INCLUDE -- F94

Used in the form:

```
INCLUDE <file>
```

Parse and load the file <name>, that can be given with the whole absolute or relative path if located in a different directory. If the name contains spaces, the name must be enclosed in double quotes:

```
INCLUDE "<file with spaces>"
```

It is the equivalent command to `LOAD`, with the difference that `INCLUDE` uses files of the Files System, while `LOAD` uses blocks (also located in the Files System, but not perceivable as files to the user).

[58] See the related word `BUFFERED` in the library `qlibrary.ft`, and study how it is implemented. `IMMEDIATE` could be rewritten in Forth in a similar way with 1 (the precedence bit) as the bit selector.

INFO n -- QFT

Print some data about qartus settings and characteristics, in the following sections:

- *Running mode*, specifying if qartus works in Forth-79 mode, the default, or in fig-Forth emulation mode.
- *Memory settings*, with all the width and starting addresses of qartus's memory sectors (dictionary, buffers, stacks); also, the current free memory amount in bytes is reported.
- *Interpreter status*, with news about last compiled word, last accessed block, last accessed buffer and the total number of compiled words.[59]
- *Execution limits*, with the extents limits for strings, numbers, the PAD, the division type, the characters width and so on.[60]
- *System dictionary status*, with the list and count of the following words classes: immediate words, compile-only words and executable-only words.
- *User dictionary status*, with the list of user words and user vocabularies.
- *List of system directories and files.*

The number n on top of stack drives the printing features:

* If n = 0 print in screens, after each you are required to type ENTER to continue, or the process is stopped.
* If n = 1 print continuously
* If n = 2 print only the sections *Running mode* and *Memory settings*
* If n = 3 print only the section *Interpreter status* (accessed also by WHERE)
* If n = 4 print only the section *Execution limits*
* If n = 5 print only the section *System dictionary status*
* If n = 6 print only the section *User dictionary status*
* If n = 7 print only the section *System directories and files*

Any other value of n is ignored.

The INFO word is only informative, but can be of great help when you want to program qartus beyond the limit.

INTERPRET -- RWS

Begin interpretation at the character indexed by the contents of >IN relative to the block number contained in BLK, continuing until the input stream is exhausted. If BLK contains zero, begin interpretation of characters from the terminal input buffer, resetting >IN.[61]

If the word name cannot be found after a search of CONTEXT and then FORTH dictionaries, it is converted to a number according to the current base. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. That also failing, an error message echoing the name with a " ?" will be given. See also "Error management".

[59] This is the output of the word WHERE, which can be found as a word by itself, belonging to the Reference Word Set of Forth-79.

[60] This is roughly what is required for the ANSI 1994 word ENVIRONMENT?, which is not part of current version of qartus.

[61] The Standard does not specify the resetting of >IN but it seems a natural choice.

IP1 IP2 IP3 IP4 U QFT

IP1, IP2, IP3 and IP4 are user variables holding the Internet Protocol 4 numbers address; by default they hold respectively 127, 0, 0 and 1 (PORT opens by default the local 127.0.0.1 address, but the programmer can choose another address by specifying new values for these user variables.

They are normal 16 bits variables, but when used as IP addresses, only the low part is used (that is the values in the range 0÷255).

J -- n C F79

Return the index of the next outer loop. May only be used within a nested DO..LOOP in the form:

```
DO ... DO ... J ... LOOP ... LOOP
```

KEY -- c F79

Leave the ASCII value of the next available character from the current input device. The typed character is not echoed, but its ASCII code is left on the stack as a single number with 8 higher bits set to zero. This procedure is blocking.

L/SCR -- 16 COM

A constant leaving the number of lines per screen. Lines are numbered from 0 to 15, thus from 0 to L/SCR-1.

LAST -- addr U RWS

A variable containing the address of the beginning of the last dictionary entry made, which may not yet be a complete or valid entry. Being an address, its range is {0..65535} as an unsigned number.

LEAVE C F79

Force termination of a DO..LOOP at the next LOOP or +LOOP by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until the loop terminating word is encountered.

LIB -- QFT

Used in the form:

```
LIB <text>
```

Set <text> (which must be a file name) as the current library.

If <text> is empty (LIB is invoked with no arguments), the library file[62] is reset to the default one ($HOME/.qartus/qlibrary.ft for Linux, Unix or Cygwin, and C:\Users\<uname>\AppData\Roaming\qartus\qlibrary.ft for the Windows© Prompt).

[62] I'm very grateful to Bruce Axtens who wrote the routine for setting the default library file under the execution of the --config option.

LICENSE -- QFT

Print the licensing text.

LIMIT FIG

A constant leaving the address just above the highest memory available for a screen buffer. In qartus this corresponds to the Data Stack start (and incidentally has the same value of S0).

LIST un -- F79

List the ASCII symbolic contents of block un (which corresponds to a block number), setting SCR to contain un. un is unsigned (as in the original Forth-79 Standard)[63]. LIST uses BLOCK to load the block in memory, in the first available screen buffer, whose value is yield by BFR. At all effects, LIST updates SCR and BFR.

LISTEN un1 -- un2 QFT

Listen to port un1 (which must be opened with ON PORT), and leave the received byte un2 (which is a value between 1 and 255); in case un2 = 0 it may mean an error in reading.

LITERAL n -- I,C F79

If compiling, then compile the stack value n as 16-bit literal, which when later executed, will leave n on the stack.[64]

LOAD un -- F79

Begin interpretation of screen un (unsigned[65]) by making it the input stream; preserve the locators of the present input stream (>IN and BLK) and the buffers, setting temporarily BLK to un and >IN to zero. If interpretation is not terminated explicitly, it will be terminated when the input stream is exhausted. Control then returns to the input stream containing LOAD, determined by the original input stream locators >IN and BLK. At all effects, LOAD sets BFR and preserves SCR and BLK.

LOAD-SYSTEM -- f E QFT

This word is used in the form:

```
LOAD-SYSTEM <file>
```

to load the dictionary previously saved in <file> by SAVE-SYSTEM, substituting current dictionary. If <file> contains spaces, it must be enclosed in double quotes.

The returned value is 0 if loading was correct, and 1 in case <file> couldn't be loaded.

[63] See also what is written for LOAD.

[64] The Standard does not specify the compile-only status for LITERAL, but the definition seems to suggest this status. Read the paragraph "A word about LITERAL (and DLITERAL)", where the issue is explained in full. This is a minor difference with the Standard (actually more a regularization rather than an enhancement).

[65] In the Forth-79 Standard, LIST explicitely uses unsigned values; for LOAD, it's not specified if the value is signed or unsigned. qartus treats consistently the both of them, so that block references are unsigned for LOAD too (as it is also for BLOCK and BUFFER); this guarantees that blocks from 1 to 65535 can be created in the File System (remember that qartus is a UNIX tool).

LOOP -- I,C F79

Increment the DO..LOOP index by one, terminating the loop if the new index is equal to or greater than the limit. The limit and index are signed integers.

At compile-time, LOOP compiles the run-time word (LOOP) and the branch address left on the control flow stack by DO.

MAX n1 n2 -- n3 "max" F79

Leave n3, the greater of the two numbers n1 and n2.

MESSAGE n -- FIG

Print the error message corresponding to code n. WARNING is first examined. If = 1, the text of line n, relative to the most recently loaded screen buffer, is printed, with n in the range {0..15}. In all other cases, the system error message n is printed, with n in the range established in errors.h. In case n is not in the defined ranges, an error condition exist. See also the chapter "Error management".

In the library, you can see for instance the word PERRORS (that is "Print ERRORS"), that uses MESSAGE to print all errors, cycling in the list using ERNUM.

MIN n1 n2 -- n3 "min" F79

Leave n3, the smaller of the two numbers n1 and n2.

MOD n1 n2 -- n3 "mod" F79

Divide n1 by n2, leaving the remainder n3, with the same sign as n1.

MOVE addr1 addr2 n -- F79

Move the specified quantity n of numbers (each contained in one cell - or two bytes) beginning at addr1 to memory at addr2. The contents of addr1 is moved first, proceeding toward high memory. If n is zero or negative, nothing is moved.

NEGATE n -- -n F79

Leave the two's complement of number n, i.e., the difference of zero less n.

NOOP -- n QFT

A constant leaving the opcode of NOOP; useful for detecting CREATE objects. See for instance the PFA word in qlibrary.ft.

NOT f1 -- f2 F79

Reverse the boolean value of flag f1. This is identical to 0= (and in effect it **is** 0=)[66].

[66] The Forth-83 Standard (and later) changed the NOT behavior to a more strict bits inversion (the one's complement); this was the main reason (or the main effect) of the turning the True value to -1, because it is the inversion of zero, that is all bits set).

OF n1 n2 -- n1 F94

At compile-time, place a forward reference of the control stack, marking the beginning of a conditional branch. At run-time, perform the test by comparing the original value and the current comparison value; if test is true, continue execution, or else jump to the next OF or default structure.

At compile time, OF compiles the run-time word (OF) and reserves space for a 32-bit address which will be later filled by ENDOF.

OFFSET -- addr U RWS

Leave the address of a variable whose content is interpreted as a block offset (with respect to the default value returned by BLOCK, which is the position of the first character in the screen buffer). The value in OFFSET is always treated as an unsigned number, with range {0..65535}.[67] OFFSET influences LOAD, that skips OFFSET characters of the screen block during interpretation of the block.

This word has not the same meaning of the OFFSET word in the Reference Word Set, which was meant to add an offset to the block number left by BLOCK, to determine the actual physical block number.

OPEN-FILE addr n fam -- file-id 0 F94
addr n fam -- 0 1

Open the file whose name is given by the character string at addr with length n, with file access fam. If the file does not exist, and W/O is specified as access mode, the file is created empty. Set the file position to the start of file if reading, or the to end of file if writing. Return values are in order:
If opening was successful, file-id is a non-zero unsigned integer in the range {1..6}; otherwise return zero.
If a connection could be established return zero, otherwise return one.

OR n1 n2 -- n3 F79

Leave the bitwise logical inclusive-or n3 of n1 and n2.

ORDER -- F94

Display the names of the vocabularies in the search list, in their present search order sequence and, afterwards, the vocabulary in which new definitions will be placed (the CURRENT vocabulary).

OUT -- addr U FIG

Leave the address of a variable holding the current output position on the terminal. It is incremented by all printing words EMIT, ." (dot-quote), SPACE, SPACES, U.R, D., D.R, U., UD., . (dot), TYPE and it's set to zero by CR, PAGE and before each command invocation from console. The word GOTOXY in qlibrary.ft sets OUT to the value of the x coordinate. The user may alter and examine OUT to control display formatting.

OVER n1 n2 -- n1 n2 n1 F79

Leave a copy of the second number on the stack.

[67] **Warning**: since OFFSET is related to a screen block, whose dimension is 1024 bytes ranging from 0 to 1023, the proper values for OFFSET are in the range {0..1023}; greater values may give bad results or cause an illegal address error.

PAD `-- addr` F79

Leave the address of a temporary scratch area, used to hold character strings for intermediate processing. The capacity of `PAD` is 84 characters[68] and is located always 200 cells after `HERE`.

PAGE `--` RWS

Clear the terminal screen, and position cursor on the upper left screen at coordinates 0,0.

The screen clearing uses VT100 codes that should apply to all monitors, included the monitors emulated in Desktop applications (for instance, it works on Linux Terminals).

PC `-- addr` U QFT

Leave the address of a variable containing the current Program Counter, which is not null during a program execution. `PC` refers to the dictionary position currently in execution, and can be altered to direct the next execution elsewhere (vectored execution). Being an address, its range is {0..65535}.

If `PC` is set to to 0 (zero), program execution will stop and return control to console. This trick can be used inside `DO..LOOP` cycles, where `EXIT` cannot be used.

`PC` is updated, after every token turn, by 2 units. All instructions involving more than two bytes, update their own count by themselves, but if you want to set the Program Counter before those instructions, you have to record PC at the proper point or calculate by yourself the real offset. Take a look at the Appendix for notes about compilation formats.[69]

PICK `n1 -- n2` F79

Return the contents of the `n1-th` stack value, not counting `n1` itself. An error condition results for `n1` less than one.

```
2 PICK is equivalent to OVER.
1 PICK is equivalent to DUP.
```

The range of `n1` is {1..n}, where n = `DEPTH`.[70]

[68] The standard prescribes 64 bytes as the minimum quantity. qartus has a greater quantity, but you can anyway increase the pad length (only increasing) using the console option `--pad=<n>`. See the man page for details.

[69] Look also at the file `stepdown.ft`, included in the package, to see how qartus uses `PC` to alter the program flow. Please, remember that, in qartus, the current execution address is NOT saved to the Return Stack but saved in `PC`.

[70] The Forth-83 Standard (and later) changed the index `n1` reference from {1..n} to {0..n}, so that the two versions are not compatible.

PORT `n1 1 -- f` QFT
`n2 0 -- -1`

With the first form, open port `n1` if a true value is on top of stack, and leave flag zero on success, or -1 if port could not be opened (i.e. socket could not be created) or -2 if connection couldn't be established.

With the second form (zero on top), close port `n2` and leave -1 to signal the end of procedure.

A warning: ports may be available or not, on your computer, depending on the firewall or/and on your network setup. So, to make qartus be aware of ports, you should enable them, otherwise the connection will always fail.

A typical usage is:

```
13 ON PORT
....  DO SOMETHING ON PORT 13...
13 OFF PORT
```

QUERY `--` F79

Accept input of up to 80 characters (or until a 'return') from the terminal, into the terminal input buffer. `WORD` may be used to accept text from this buffer as the input stream, by setting `>IN` and `BLK` to zero. Query has a limited keyboard control (echo for the current key-press, Backspace and Return).

QUIT `--` F79

Clear the return stack, setting execution mode, and return control to the terminal. No message is given.

R0 `-- addr` `"r-zero"` FIG

A constant leaving the address of the bottom of the Return Stack.

R> `-- n` C `"r-from"` F79

Remove `n` from the return stack to the data stack.

R@ `-- n` C `"r-fetch"` F79

Copy the number on top of the return stack to the data stack.

R/O `-- fam` `"ar-slash-oh"` F94

A constant which returns the Read File Access Method `fam` = 1.

READ-LINE `addr n1 file-id -- n2 1 0` F94
`addr n1 file-id -- 0 0 1`
`addr n1 file-id -- 0 0 2`

Read one line of text from the file identified by `file-id` and store it at address `addr`, for at most `n1` characters. Terminate the reading if a line terminator was found, or `n1` characters were read; the return values are in order:
- if the reading was successful, `n2` is the actual count of read characters; otherwise return zero;
- in case of successful reading, if the end of file was reached after at least one `READ-LINE` execution, return one, otherwise return zero.
- in case of successful reading, return zero; in case something went wrong, return one in case the file was never been opened, and two in case the file connection wasn't correctly initiated.

REPEAT I,C F79

Used in a colon-definition in the form:

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

At run-time, `REPEAT` returns to just after the corresponding `BEGIN`.

At compile time, `REPEAT` compiles the run-time word `(REPEAT)`, reserves one 32-bit cell, retrieves two addresses from the control flow stack, the topmost left by `WHILE` and the innermost by `BEGIN`; compiles its address on the address left by `WHILE`, and compiles the address of `BEGIN` on the reserved cell.

REQUIRE E QFT

Used in the form:

```
REQUIRE <name>
```

Parse file `qlibrary.ft`[71] (looking for it in the default library file), read and compile the word `<name>` and all dependencies word that are in the same library[72]. At all effects, it's like the definition of `<name>` is typed from keyboard, along with all the other definitions that occur in it. `REQUIRE` cannot be compiled, and is executable-only.

This word works like `ENSURE`, with the difference that `REQUIRE` will always load the word, existing or not in the dictionary, while `ENSURE` will do it only if the word does not exist in the dictionary.

[71] Please note that a) since the reading of the definition starts from the `: <name>` token and proceeds until the next colon is met, all intermediate direct Forth code is executed (included `VARIABLE`, `IMMEDIATE` etc...); b) only colon definitions are loaded through `REQUIRE`; c) constants in the library are defined as colon definitions to let `REQUIRE` find them; for what about all the library variables, you have to declare them on your own before requiring the words that use them (e.g. `TIMING` - see ahead), or define them through `CREATE`. Since the library is written in `DECIMAL`, `REQUIRE` (and `ENSURE`) compile the word in `DECIMAL`, restoring current `BASE` afterwards. So that you must not care of setting `DECIMAL` before loading a library word.
If you plan to enhance `qlibrary.ft` by adding variables, constants or `CREATE`d objects outside definitions, remember to group them and precede them with a dummy definition; when this dummy definition is `REQUIRE`d, all the variables/constants/`CREATE` words that follow will be added to the dictionary; otherwise, they must be explicitly typed from keyboard or written to file/block. See for instance the `TIMER` words in the library.

[72] Actually, the file `qlibrary.ft`, a collection of words in form of colon definitions without unresolved dependencies.

ROLL `n1 –` F79

Extract the `n1`th stack value to the top of the stack, not counting `n1` itself, moving the remaining values into the vacated position. An error condition results for `n1` less than one.

```
3 ROLL = ROT
2 ROLL = SWAP
1 ROLL = null operation
```

The range of `n1` is {1..n}, where n = `DEPTH`.[73]

ROT `n1 n2 n3 -- n2 n3 n1 "rote"` F79

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

S0 `-- addr "s-zero"` RWS

A constant leaving the address of the bottom of the Data Stack (and incidentally this is the same value returned by `LIMIT`).

SAVE-BUFFERS F79

Write all blocks to mass-storage that have been marked as `UPDATE`d. An error condition results if mass-storage writing is not possible.

SAVE-SYSTEM `-- f E` QFT

This word is used in the form:

```
SAVE-SYSTEM <file>
```

to save the current dictionary in `<file>`; the file will contain the entire dictionary (variables, compiled words, constants), but will not include the stacks or the buffers. `<file>` can be lately loaded in memory with `LOAD-SYSTEM`. If `<file>` contains spaces, it must be enclosed in double quotes. This word is execute-only (cannot be compiled).

The returned value is 0 if saving could get accomplished, and 1 in case of error on file opening or writing.

SCR `-- addr U` F79

Leave the address of a variable containing the number of the screen most recently listed with `LIST` or edited through `EDIT`.

[73] The Forth-83 Standard (and later) changed the index range from {1..n} to {0..n}, so that the two versions are not directly compatible.

SEARCH E QFT

Used in the form:

```
SEARCH <text>
```

Search in every block for the string <text>, and print every block number, and the line where it occurs enclosed in parentheses.

The search is case insensitive (that is, if FORTH is the search pattern, FORTH, forth and Forth will all match the search).

SEARCHLIB E QFT

Used in the form:

```
SEARCHLIB <text>
```

List all the colon definitions in the current library that contain the pattern <text>; at the end, ok is printed.

If the pattern could not be found, nothing is printed.

If the pattern <text> is empty (not given), print the names of all words in the library, á-la-WORDS.

The search is case insensitive (that is, if FORTH is the search pattern, FORTH, forth and Forth will all match the search).

SEE -- COM

Used in the following form:

```
SEE <name>
```

Decompile <name> to see its structure; the output is formatted and does not maintain the original spacing. Numbers are printed according to BASE. Built-in words cannot be decompiled. If <name> is not found in the CURRENT dictionary, a condition of error exists.

Vocabulary entries show the (vocabulary) token, followed by the vocabulary code in curly brackets.

Isolated bytes in the range {0..255} are printed as bytes numbers in current BASE, followed by their graphic representation enclosed in curly brackets if in the range {33..255}, or followed by a blank enclosed in curly brackets if in the range {0..32}. Normally, isolated bytes signal incorrect compilation or incomplete features in qartus: in this last case, write me.

SEND un1 un2 -- QFT

Send byte un1 to port un2 (which must be opened with ON PORT); leave no return value.

SHIFT `n1 n2 -- n3` RWS

Logical shift of `n1` leftward `n2` bits if `n2` in positive, rightward if `n2` is negative. Zeros are shifted into vacated bit positions.

SIGN `n --` F79

Insert the ASCII "−" (minus sign) into the pictured numeric output string, if `n` is negative. Used only between <# and #>.[74]

SPACE `--` F79

Transmit an ASCII blank to the terminal screen.

SPACES `n --` F79

Transmit `n` spaces to the terminal screen. Take no action for `n` = zero or less.

SPAN `-- addr` U F83

Leave the address of a variable containing the number of characters typed during last `EXPECT` execution, in the range {0..255} as an unsigned byte.

STATE `-- addr` U F79

Leave the address of a variable containing the compilation state, which is non-zero for the compilation state, and 0 (zero) for the interpretation state.

STDERR `-- n` QFT

A constant leaving the code for the standard output channel `stderr` (the UNIX standard error channel) that may be used with `WRITE-LINE`.

STDIN `-- n` QFT

A constant leaving the code for the standard input channel `stdin` (the UNIX standard input channel) that may be used with `READ-LINE`.

STDOUT `-- n` QFT

A constant leaving the code for the standard output channel `stdout` (the UNIX standard output channel) that may be used with `WRITE-LINE`.

SWAP `n1 n2 -- n2 n1` F79

Exchange the top two stack values.

[74] It's curious noting that the Forth-79 Standard set this word as compile-only. This seems a typo, rather that a misconception; so in qartus `SIGN` is not compile-only; it has nonetheless the same behaviour of `HOLD`, that is, it can be used only between <# and #>. This is a minor deviation (a regularization, actually) from the Standard.

TASK FIG

A no-operation word[75] which can be used as safe null operation (for instance in referenced functions, to reset some feature to null, avoiding dangerous zeroes).
Originally, in fig-Forth, TASK was used to mark the boundary between applications. By forgetting TASK and re-compiling, an application could be discarded in its entirety. qartus does not need such a feature, because of its peculiar FORGET behaviour.

THEN -- I,C F79

Used in a colon-definition in the form:

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

THEN is the point where execution resumes after ELSE or IF (when no ELSE is present).

At compile time, THEN compiles the run-time word (THEN), retrieving from the control flow stack the address left there by IF or ELSE, and compiling there its proper address.

TIB -- addr FIG

A constant leaving the address of the first character of the Terminal Input Buffer.

TIME -- n1 n2 n3 QFT

Leave on stack current seconds n1, current minutes n2 and current hor n3 on top. n1 is in the range {0..59}, n2 is in the range {0..59}, n3 in the range {0..23}.

TRACE E QFT

Used in the form:

```
TRACE <name>
```

Enable the debugger on the user word <name>, by setting a tracepoint at it; the debugger exits when <name> has ended its duty. The tracepoint is not removed until the end of session, or until another different word is TRACEd. To manually disable the tracepoint, use the parameter-less version:

```
TRACE
```

TRACE is also disabled by OFF DEBUG (see).

Built-in words cannot be TRACEd, or an error condition exists.

TRUE -- 1 COM

A constant leaving the true value.

[75] I underline here that a no-operation word does nothing, has no arguments, and returns nothing.

TYPE addr n -- F79

Transmit n characters beginning at address addr to the terminal. No action takes place for n less than or equal to zero. If the zero string terminator is found before the nth character, stop.[76]

U* un1 un2 -- ud3 "u-times" F79

Perform an unsigned multiplication of un1 by un2, leaving the double unsigned number product ud3. All values are unsigned.

U. un -- "u-dot" F79

Display un converted according to BASE as an unsigned number, in a free field format, with one trailing blank following.

U.R un n -- "u-dot-r" RWS

Display un converted according to BASE, right aligned in an n-characters wide field. If n is negative or smaller than the characters required for un, no leading spaces are given.

U/MOD ud1 un2 -- un3 un4 "u-divide-mod" F79

Perform the unsigned division of double number ud1 by un2, leaving the remainder un3, and the quotient un4. All values are unsigned.

U< un1 un2 -- f "u-less-than" F79

Leave the flag f representing the magnitude comparison of un1 < un2 where un1 and un2 are treated as 16-bit unsigned integers.

UD. ud -- "u-d-dot" COM

Display ud converted according to BASE as an unsigned double number, in a free field format, with one trailing blank following.

UD/ ud1 ud2 -- ud3 "u-d-slash" COM

Divide the double number ud1 by the double ud2 and leave the double quotient ud3. All values are unsigned. See / (division).

UNTIL f -- I,C F79

Within a colon-definition, mark the end of a BEGIN..UNTIL loop, which will terminate based on flag f. If f is true, the loop is terminated. If f is false, execution returns to the first word after BEGIN. BEGIN..UNTIL structures may be nested. It may be spelled also as END with identical functions.

At compile time, UNTIL or END compile the run-time word (UNTIL), and reserve one 32-bit unit to store the address left on the control flow stack by BEGIN.

[76] I don't know if this is a qartus's feature or if this is common to all Forth-79 interpreters and compilers. I only know that this is an useful feature.

UPDATE -- F79

Mark the most recently referenced block as modified. The block will subsequently be automatically transferred to mass storage, should its memory buffer be needed for storage of a different block, or upon execution of SAVE-BUFFERS.

USE -- QFT

Used in the form:

```
USE <text>
```

Set <text> (which must be a legal directory path) as the current directory where to search for blocks. The directory is created empty in case it is not found.

If <text> is a relative path, it will be created in current directory; if <text> is empty (USE is invoked with no arguments), the system directory[77] is reset to the default one (`$HOME/.qartus/blocks/` for Linux, Unix and Cygwin, and `C:\Users\<uname>\AppData\Roaming\qartus\blocks\` for the Windows© Prompt).

VARIABLE -- F79

A defining word executed in the form:

```
VARIABLE <name>
```

to create a dictionary entry for <name> and allot two bytes (one cell) for storage in the parameter field, setting the variable contents to zero. When <name> is later executed, it will place the starting storage address on the stack.

VERSION -- n1 n2 QFT

Leave on the stack two 16-bit numbers: the minor version number n1 and the major version number n2 (on top), for versioning checking of the user programs.

VOCABULARY -- F79

A defining word executed in the form:

```
VOCABULARY <name>
```

to create (in the CURRENT vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent execution of <name> will make it the CONTEXT vocabulary. When <name> becomes the CURRENT vocabulary (see DEFINITIONS), new definitions will be created in that vocabulary.

In qartus <name> will include all definitions of the vocabulary in which <name> is itself defined. All vocabularies ultimately chain to FORTH. By convention, vocabulary names are to be declared IMMEDIATE.[78]

---

[77] I'm very grateful to Bruce Axtens who wrote the routine for setting the default blocks directory under the execution of the `--config` option.

[78] The behaviour is roughly similar to the fig-Forth model. This is a difference with the Standard. See also the chapter "VOCABULARY problems".

W/O -- fam "w-oh" F94

A constant which returns the Write File Access Method fam = 2.

W0 -- addr "w-zero" YTH

A constant leaving the address of the bottom of the WORD buffer.

WARNING -- addr FIG

Leave the address of a variable containing a value controlling ERROR messages output. If = 0 (default case), messages will be retrieved from the default errors strings array and the default error message management is simulated. If WARNING = 1, and at least one block was loaded, strings will be retrieved from the most recent screen buffer (ruled by the BFR variable that you can control). If WARNING = -1, (ABORT) is called, for a user specific procedure. After their completion, all cases are followed by the default error resolution operations. See also the entry for ERROR in this manual and the chapter "Error management".

WHERE -- RWS

Print information about the status of qartus, (e.g., after an error abort), printing[79]:

- the last compiled word name
- the last accessed block number
- the last accessed buffer number
- the total compiled words number

WHILE f -- I,C F79

Used in the form:

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

where f is a flag. Select conditional execution based on flag f. On a true f, continue execution through to REPEAT, which then returns back to just after BEGIN. On a false f, skip execution to just after REPEAT, exiting the structure.

At compile time, WHILE compiles the run-time word (WHILE), push its address on the control flow stack and reserves one 32-bit unit.

WIDTH -- addr U FIG

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It's by default set to 31 and has a (theoretical) range {1..255}. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.[80]

I want to underline here anyway that FORTH and EDITOR, the default vocabularies, are IMMEDIATE words, as required by the Standard (to compile the reference for those, [COMPILE] must be used), and this is an indirect confirmation that vocabularies are to be declared IMMEDIATE.

[79] This information, along with many others, is also printed by INFO.

[80] **Watch out!** An improper name length may cause irregular functioning or memory segmentation errors; if the length is fixed to a value lesser than the maximum value already compiled (in a user/system word), the system may crash! Change this variable only if strictly necessary, preferably increasing it up to and not beyond 255 (and never decreasing it less than 31).

WORD `c -- addr` F79

Receive[81] characters from the input stream according to the delimiter character `c` and place the characters in a string beginning at `caddr+1` (in `WORD`'s buffer). If `c=1`, receives all characters from the input stream. The characters count is placed in the byte position at `addr`. An error condition results if `c` is the ASCII null or if the count exceeds 255. Initial occurrences of `c` in the input stream are ignored only when they are blanks[82]. If `c` appears in the input stream as a terminating character, it is appended to the string but not included in the count. If the input stream is exhausted before `c` is encountered as a terminating character, the terminating null character is appended instead of `c`. A zero length will result if the input stream is empty when `WORD` is called.[83]

WORDS `--` COM

List all the definition names belonging to the list presented by `ORDER`, that is from the `CONTEXT` vocabulary down to `FORTH`, in the created chain. As said for `VOCABULARY`, the `FORTH` word-list is always active, because all dictionaries are chained to it.[84]

WRITE-LINE `addr n file-id -- 0` F94
`addr n file-id -- 1`

Write at most `n` characters from `addr` to the file identified by `file-id`, adding a line terminator after `n` characters or after the natural character-list end. Return zero on success, one otherwise.

XOR `n1 n2 -- n3` `"x-or"` F79

Leave the bitwise exclusive-or of two numbers.

[`--` I `"left-bracket"` F79

Stop the compilation mode. The words after `[` are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with `]`. See `]` and `COMPILE`.

---

[81] This definition is not the one contained in the Standard, but a mix of the one proposed by Robert L. Smith in his "*Forth Standards corner*", about the `WORD` procedure (contained in *Forth Dimensions* vol. III, N. 3, September/October 1981) mixed with the one designed by me. This new definition, while does not change the Standard effect of `WORD`, tries to resolve some problems in the Standard definition: no error condition specified in case the delimiter is null; unclear definition of resulting packed string length; unclear text in case of empty stream. And since, in according to Robert L. Smith observations (cit), the skipping of the leading occurrences of the delimiter prevents `WORD` from treating empty strings, please note that qartus does not skip any leading occurrences when they are not blanks. This is a minor difference with the Standard (an enhancement, actually).

[82] This is a deviation from the Standard.

[83] In the Standard it's not clear if `WORD` has a dedicated buffer or if it uses a volatile memory portion (like `PAD`). I built qartus assigning to `WORD` a proper and independent buffer, to avoid collisions between the `WORD` buffer and the dictionary. This buffer is 1024 bytes wide, and is split into four segments, active one at a time. Each segment is 256 wide (255 characters + 1 terminator) and its address is returned in `addr`, so that you can compare two strings got through `WORD` (because it always returns two distinct addresses in two consecutive calls, or even three or four).

[84] See for contrast the `VLIST` word, in `qlibrary.ft`, that lists all definitions of the `CONTEXT` dictionary only.

[COMPILE] I,C "bracket-compile" F79

Used in a colon-definition in the form:

```
[COMPILE] <name>
```

Force compilation of the following word. This, for instance, allows compilation of an IMMEDIATE word when it would otherwise be executed. In case the word EXEC-ONLY is applied to a word (to signal it must not be directly compiled), [COMPILE] can override this bit state and compile the word that follows.

**** -- "slash" COM

Ignore all text from the character following the space that follows to the end of the input line in any regular file, and to the end of line in a block (this means that this comment extend until the last character of current line in a screen 16 x 64 characters - see LOAD)

] -- "right-bracket" F79

Set the compilation mode. The text from the input stream is subsequently compiled. See [.

3.3. The Line Editor in detail

The Forth-79 Standard does not specify commands for the line editor, or what must be the editor like. It only specifies (in the Reference Word Set) the EDITOR vocabulary, as if suggesting that all possible editing words must be put in there, without saying what these words should be.

So the editor commands added to qartus are mainly those described in "EXTENDED fig-Forth", by Patrick L. Mullarky, dated 1981. Some additions were also grabbed from "Forth On The BBC Microcomputer", by Richard De Grandis Harrison, dated January 17, 1983, published by Acornsoft.

The editor words are available only after EDITOR has been typed. If you often forget this (as I do), define LIST to enable editor by default:

```
: LIST   LIST  EDITOR ;
```

A word expecting a list of characters <text> must be used as the last word on the command line; words expecting only numbers on stack or acting without arguments or parameters can be cumulated on the command line.

CLEAR `n --`

Clear screen n (with n = 1,2 or 3) by filling it with blanks (ASCII 32), and destroying any previous information.

COPY `n1 n2 --`

Copy screen n1 onto screen n2 (with n1 and n2 = 1,2 or 3). It overwrites any previous content of screen n2. This operation does not change the current buffer screen references (default buffer and cursor position).

MARK

Mark the current screen as modified. A subsequent SAVE-BUFFERS command will cause the entire screen to be written out. It's a synonym of UPDATE.

TEXT

Wait for the input of text from keyboard, up to 64 characters (or until the Return key is pressed) and put them into PAD. Text may be used for example by C or I to insert it in a specific line of the screen buffer.[85] The remaining part of PAD, not affected by TEXT, is erased each time.

[85] This command, specifically targeted for the Line Editor, is not the equivalent to the command TEXT in the Reference Words Section; in particular, this word does not use EXPECT to get the string into PAD and thus SPAN is not affected by this word. If you should need SPAN, you can rewrite TEXT in the EDITOR vocabulary like this:

```
EDITOR DEFINITIONS
: TEXT    PAD 64 EXPECT
```

with identical effects and with SPAN containing the characters actually typed by the user.

TILL

Used in the form:

```
TILL <text>
```

Delete all text from the current cursor position until end of the matching string <text>. The deletion involves only the current line, and does not act on the rest of the screen.

TOP

Position the cursor at the upper left corner of the current screen (start of screen at row=0 and line=0, or position=0 and cursor=0)

STAMP

Substitute line 0 with a stamp line with date and initials, that must be successively integrated by the programmer.

A

Used in the form

```
A "cccc1" "cccc2"
```

Look for character string cccc1 in the current screen, starting from the current cursor position, then changes the first occurrence of cccc1 with cccc2, saving cccc1 to PAD for another search with N. Patterns must always be included in double quotes.

B

Move the cursor backward for as many characters are in PAD, until the first null character. The cursor movement is performed by an inner call to M.

C

Used mainly in the two forms:

```
C <text>
C
```

Insert text into the current line at the current cursor position. If the argument <text> is given (first form), use <text> as the insertion text; if the second form without argument is used, use the text in PAD as the insertion text.

D `n --`

Delete line n in the current screen and moves up all following lines, creating a new (empty) 15th line. This word also saves the contents of the deleted line into PAD, so that you can use the editor word I later, if needed.

E n --

Erase line n in the current screen, i.e. filling it with blanks, without moving up the following line.

F

Used in the form

F <text>

Look for character string <text> in the current screen, starting from the current cursor position. This words also saves the string <text> in PAD.

H n --

Copy the content of line n to PAD, leaving the line untouched. It is equivalent to the sequence

```
n D
n I
```

I n --

Insert the content of PAD in a new line created immediately above line n in the current screen, and then moves all following lines, including old nth, down one line (15th line is lost).

L

List current screen. The current screen is changed by n LIST which will list out screen n and make it the current screen. The current screen may also be changed by setting variable BFR.

M n --

Move the cursor n characters forward or backward (backward if n is negative).

N

Look for next occurrence of the text in PAD. If this text was left by F, N can be used as a faster second instance of F for the same search pattern.

P n --

It is used in the form

n P <text>

Put the character string <text> into line n, and erase the previous content, if any. Only one space after P is discarded, so that <text> may contain even leading blanks.

R n --

Replace the content of line in the current screen with the text in PAD.

S n --

Spread current screen at line n, originating a new empty line immediately preceding line n, and moving all following lines down one line (15th is lost).

T n --

Type out line n of current screen; the cursor position is pointed by a caret (^) put under the correspondent character in case it belongs to line n. This word is called by all the editor words that alter the text and print the modified or search line.

X

It is used in the form

X <text>

Extract the first character string occurrence of <text> and shorten up the line. The extracted text is copied to PAD. This command, as F, begins the search from the current cursor position.

To these, the following variables (belonging to the FORTH vocabulary) were added, to ease the user programming the screen buffers (with the note that screens are numbered 0 to 15 and positions 0 to 63):

#L U

The value of the current editing line in the current editing buffer. It is updated automatically by the various editor commands. The range is {0..15}.

#P U

The value of the position of the cursor in the current editing row. It is updated automatically by the various editor commands. The range is {0..63}.

R# U

The value of the current cursor position in current editing buffer as an absolute reference in the range {0..1023}. It is updated automatically by the various editor commands and by LOAD.

3.3.1. Some Line Editor techniques

Here is some tips for a proper usage of the Line Editor commands.

Working on the screen
List the screen: use `L`
Clean the screen: use `CLEAR`
Copy one screen on another: use `COPY`
Reset cursor position: use `TOP`
Mark screen as updated: use `MARK`

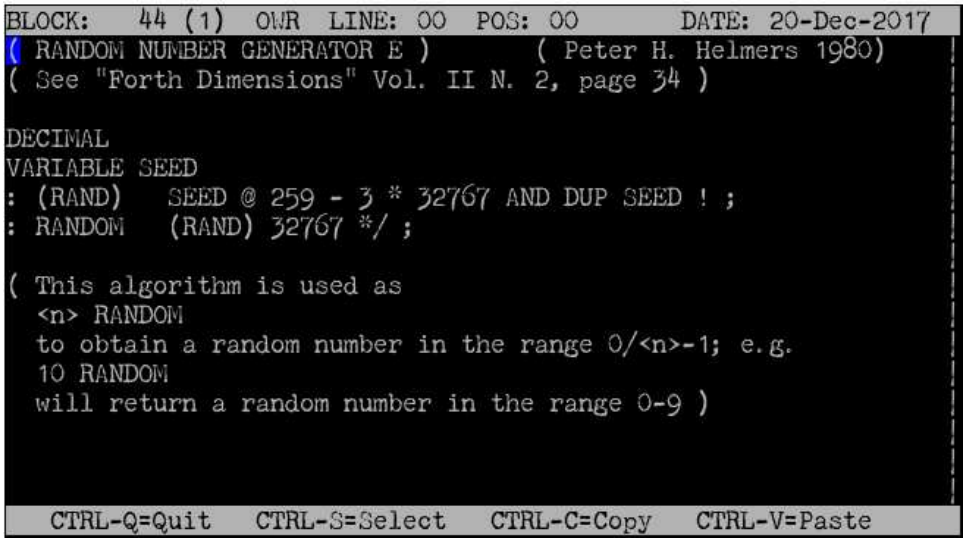
Working on a line
Show a line: use `T`
Set line text: use `P`
Clean the line: use `E`
Remove the line: use `D`
Move line to another row: use `D`; use `I`
Insert a void line: use `S`
Replace line with `PAD` content: use `R`
Copy the line to `PAD`: use `H`
Copy a line to another row: use `H`; use `R`
Insert a line: use `I`
Insert a line: use `S`; use `R` (assuming line content is in `PAD`)

Searching text and set cursor
Finding a string: use `F` (cursor set at the end of the match)
Finding next occurrence: use `N`
Setting cursor at the beginning of the match: use `F`; use `B`
Setting cursor at a specific point: use `M` (some calculation is involved)

Changing a line content
(Note: *these words depend on the cursor position; in case you have to consider the screen from the start, use* `TOP` *before any of them*)
Delete from cursor to a given pattern: use `TILL`
Delete pattern: use `X`
Substitute text: use `A old new`
Substitute text: use `X old`; use `C new`
Substitute text: use `X old`; use `TEXT` typing `new`; use `C` (no args)

3.4. The Screen Editor in detail

Following some Forth traditions (thanks to old Forth environments like F-PC, a powerful Forth for DOS, which had some very advanced instruments, like for instance the screen editor), and listening to the proper advices of tester Ian Jones about such subject, I decided to add a built-in Screen Editor, suitable for the editing of Forth blocks (1024 bytes in 16 rows of 64 characters each), invoked though the word `EDIT`, as in the following figure:



`EDIT` must find the block number to be edited on top of stack. It works in the same vein of `LOAD`. Don't expect `EDIT` to be a sort of Microsoft Word, or Libreoffice Writer, though. It's a simple tool, with many features, yes, but its most important feature is that it's an editor targeted for Forth.

`EDIT` has two states: the overwrite state (the default, safer mode), and the insertion state, each with obvious scopes. These states can be toggled with CTRL-J.

The cursor can go anywhere in the screen but not past the dominion of the 1024 characters. This helps in maintaining correctly dimensioned lines. Moreover, all places where you don't see printable characters are not really empty, but filled with blanks, so the screen **always** contains 1024 bytes: this assures that anytime you decide to save it to the storage device, no less and no more than 1024 bytes are saved. In any case, the editor can load any lines up to 16, with length up to 64 characters, terminated by the End-Of-Line character (in all, less than 1024 bytes), filling the rest with spaces; when you will save this block, it will be correctly set to 1024 bytes.

Warning: since the block is loaded in its entirety, using the last character of a line and the first character of the following line may lead to error in compilation, because the two are seen as contiguous (unless this is intended). So, you can use the last character of a line if the next line is indented, or use the first character of a line if previous line terminates with a blank.

The Screen Editor disables the standard interrupt state, to let using all commands through the CTRL key[86] (the standard state is restored when exiting). All the remaining non-printable usual keys do work (I hope for you) as expected, except TAB, which acts like CTRL-I. The copy command CTRL-C uses a private buffer, not shared with the Operating System.[87]

[86] Please note that in the Screen Editor CTRL-Z does not work here as the shell 'suspend' interrupt.

[87] On emulated terminals, you can do Copy&Paste operations with the mouse; first, moving the mouse, select the portion you want to copy (right-click/Copy); this operation can be done on any window (for instance, from a site page in the browser or the `EDIT` window itself); then you switch to the `EDIT` window, and move to the place where to paste (with the arrows, not the mouse); then you

3.4.1. Status lines

The `EDIT` screen has an upper line and a bottom line (both not editable and in reverse colours) which serve as information centers.

The top line reports the following data:

- `BLOCK:` block number (the next number in brackets is the buffer number)
- `INS/OWR` insertion/overwrite state
- `LINE: POS:` line and position of the cursor (lines are numbered 0..15 and positions 0..63)
- If the block buffer was changed after loading, the string `UPD` appears following the `POS` number; `UPD` disappears when the block is saved or mark as not updated.
- `DATE:` the current date in a common and unambiguous format: e.g. 10-Nov-2018

The bottom line reports useful data, like a brief help in normal mode, runtime messages and, when asked for input, it turns to a user typing field.[88]

Anytime, the least accessed screen is the chosen one for editing, starting from screen #1.

3.4.2. Screen Editor commands

All commands are driven by the CTRL key:

CTRL-A Select all text in the current buffer.

CTRL-B or Page Down
: Boot next block. Start editing a new buffer screen (even empty) attributing it the current block number + 1. It may also be used to 'read' several consecutive blocks in the Screen Editor.

CTRL- or Page Up
: Boot previous block. Start editing a new buffer screen (even empty) attributing it the current block number - 1. It may also be used to 'read backward' several consecutive blocks in the Screen Editor.

CTRL-C Copy selected text; if no selection was made, nothing is done.

CTRL-D Delete current line, shifting up lower lines; a 15th line is created empty.

CTRL-E Edit another block in the next available screen and set the relative buffer as the current buffer. The block number is asked on the bottom line.

CTRL-F Find a string in current screen; the matching string is asked on the bottom line. The search starts from the current cursor position. After search, the cursor position is set to the start of the matching string, if found, or remains in the original position if not found (the message "Text not found" appears on the bottom line).

CTRL-G Repeat last search made with CTRL-F, starting search on character after current cursor position (to avoid finding the same last search).

CTRL-H Show a short help screen; ESC to return to the current editing screen. Editing is disabled while the help screen is active.

---

paste (right-click/Paste). The CTRL-C and CTRL-V are not related with the right-clicks of the Operating System; that is, either you use the mouse, or you use CTRL-S/CTRL-C/_CTRL-V from keyboard; the two systems don't use the same memory buffer, but the arrows (for positioning) must be used in both cases.

[88] During input in the typing field, keep in mind that: for commands that ask for a number, only characters in the range {0..9} can be typed, for at most 5 digits; for commands that ask for a string, only ASCII characters in the range {32..126} can be typed, for at most 50 characters. Of all the other control characters, only BACKSPACE and ENTER are evaluated during the typing: the first deletes last character, the second confirms the whole input.

CTRL-I Insert an empty line, shifting down current and lower lines; 15th line is lost. The TAB key performs the same task, because it's bound to the same CTRL command.

CTRL-J Toggle the insert/overwrite state (overwrite being the default). Labels `INS` or `OWR` appear on the top line, to signal the proper active state.

CTRL-K Erase current line filling it with spaces.

CTRL-L Quit `EDIT` (the Screen Editor) and enter the Line Editor mode (enabling the `EDITOR` vocabulary and listing current buffer).

CTRL-M Equivalent to the ENTER key.

CTRL-N Clear current buffer, setting it as new and not updated. All content is lost. The clearing involves only the memory buffer, not the original file.

CTRL-O Re-open the block file, reloading all the block with the data on file, and losing all changes applied to the buffer. Use this command with care, because it will lose all your changes, though sometimes this is just the thing to do, when the text gets cluttered and we feel that the changes we've made don't satisfy us any more. Useful also when re-entering `EDIT` (in this case the buffer is already engaged, so the block is not re-loaded) and we want to re-edit from start.

CTRL-P Put a default stamp on the first line of the screen (the stamp is the same of the command `STAMP` of the line editor).

CTRL-Q Quit the editor and get back to qartus's console, without saving. **Watch out!** `EDIT` does not ask you to save an updated screen, so, if saving is your intention, remember to type CTRL-W to save the block, before quitting, or type `SAVE-BUFFERS` once back to the console.

CTRL-R Replace the latest string occurrence searched with CTRL-F with a new string; the new string is asked on the bottom line only for the first substitution (in case no search was performed, the match string is also asked, priorly, on the bottom line). The search starts from the current cursor position. After the substitution, the cursor position is set to the start of the substituted string, if found, or remains in the original position if not found (the message "Text not found" appears on the bottom line, and the new string is reset). Multiple substitutions can be applied simply repeating the CTRL-R command.

CTRL-S Toggle selection state. Selection always starts from current cursor position and is extended forward or backward, upward or downward with the arrow keys.

CTRL-T Set the cursor to Top, i.e. line 0, position 0 (the upper left corner); if the cursor is already there, put it at line 15, position 63 (the lower right corner).

CTRL-U Undo last operation. **Watch out!** There's only one undo level for all commands.

CTRL-V Paste the selected text to the current cursor position; if no text was copied, nothing is pasted.

CTRL-W Write the current editing block to disk; other buffers are not touched by this operation. Of course, the same operation can be done cumulatively on all buffers with `SAVE-BUFFERS`, after getting back to qartus's console (the message "Block file was saved to file system." appears on the bottom line). CTRL-W is not executed after a CTRL-K, which sets current buffer as updated.

CTRL-X Copy and cut the selected text; if no selection is made, nothing is done.

CTRL-Z Set current block as not updated, so that a further `SAVE-BUFFERS` command won't save the block to disk, and physically remove the file from the file system (the message "Block file was removed from file system." appears on the bottom line). It is someway equivalent to an `EMPTY-BUFFERS` directed on a single buffer. CTRL-Z disables also a following CTRL-W on the same buffer. Any editing on the buffer will reset this command (the block would be marked updated) and on a subsequent saving (with CTRL-W or with `SAVE-BUFFERS`), while the file is marked updated, the file is recreated anew.

The ENTER key moves to the beginning of the next line. If ENTER is pressed in insert mode, the text is

broken and the part following ENTER (including current cursor position and up to the end of line) is moved on next line created as new (the 15th is lost). In overwrite mode, no breaking is done.

The control Keys available on editing are:

- The arrow keys
- ESC (only for Help Screen exit)
- CANC - delete current character
- HOME - go to the first position of current line
- END - go to the last position of current line
- BACKSPACE - remove current character and shift left

Other control Keys are left to the Operating System (for instance, the Stamp key that captures the screen, present on some OS).

The Screen Editor is still experimental, and I expect some bugs. Please, if you find one, write to me immediately. Thanks.

A note for computers having a keyboard with a dedicated numerical keypad: this pad generally works in two modes: with the block-number ON or with the block-number OFF. If it is ON, the numeric keys emit their numerical value. If it is OFF, the pad works as a directional tool, with keys 8,4,6,2 as arrows, 9,3 are page-up and pagedown, 7 is used as page-top, 1 as end-of-line and 0 as INS, with the 5 key disabled. When the Screen Editor is used and the block-number is deactivated, the pad works normally, but some keys (namely 7, 6, 5), if used in combination with the CTRL key, are disabled because they interfere with the editor commands.[89]

BTW: there's an easter egg, in the editor. Can you find it? Peeping into the code is forbidden! ;-)

3.5. fig-Compatibility

The compatibility flag `FIG-STATE` is a variable that influences the behaviour of qartus with respect to the fig-Forth compatibility; if it contains zero, qartus behaves as a Forth-79 system; if is contains a number different from zero, in order to make qartus behave like fig-Forth, the following words change their behaviour:

`+LOOP` (include the lesser limit in case of negative increment)
`2VARIABLE` (use the double number on the stack as starting value)
`ABORT` (print an appropriate message, required by fig-Forth)
`I` (available also outside a `DO..LOOP` and working as `R@`)
`VARIABLE` (use the number on the stack as a starting value)
`WORD` (place the counted string to `HERE`)

The compatibility does not automatically load the missing fig-Forth words. Block 20000 contains the library reference of words related to fig-Forth; to turn your system in a (quasi) fig-Forth environment, type thus

```
1 FIG-STATE !
20000 LOAD
```

and you're done. See also `79-STANDARD`.

[89] If you use an emulated terminal on your desktop, maybe you can use the cute commands CTRL/+ and CTRL/- to expand or reduce the terminal fonts. These commands are not intercepted by the Screen Editor.

3.6. Service words

In developing qartus I often needed some service words that served to inspect the memory and the stack, the state and so forth. So, to avoid clutching the dictionary with sparse words, I added them as super-immediate words not dictionary-based. Two of these would eventually became part of qartus, due to their utility: `BYE` and `COLD`. The remaining were to be deleted, but I ultimately thought that if they were useful to me, they might be useful for you too. So I decided to maintain them, to signal also you cannot name a word with the same characters combination of these (even if I tried to use improbable names), because it won't be reached. And remember that they are not compilable, they are immediately executed when met, and there is no way to use `[COMPILE]` with them and readline does not know them, so they remain pure keyboard console tools (see anyway the note in `BYE`).

I expose them here, in the same manner of the rest of the dictionary, but remember that they are not contained in the dictionary and quite certainly they are not portable.

`@STAT` S QFT

Enable the statistics to be printed at the end of the session. It's equivalent to the option `--statistics` typed at start.

`CF@` S QFT

Print the control flow stack content in a non-destructive way. In normal usage, it should always show an empty control flow stack. I built it to check when built-in words under development didn't correctly work, leaving some scattered values back.

`CHECK-DICT` S QFT

Try to fix the database in case it gets damaged. **Watch out!** This command tries to repair the database, starting from the bottom and proceeding upward, by removing the first item which does not match the link addresses references with the previous word; `HERE` and `LAST` are re-addressed in case. This normally implies the loss of some user words (the faulty one and all that follow). But if the fault is found in the built-in section (due to improper memory writing), using `CHECK-DICT` can cause the deletion of some built-in words. So, in case of cluttered and faulty dictionary, use `CHECK-DICT`, but if the system remains unresponsive, type `COLD` and restart.

`DS` `-- addr` S QFT

Leave the memory position of the next free unit of the top of stack; the last value was stored in `DS -2` and can be checked with:

```
DS 2- @ .
```

I built it to test the stack and the stack-based words without using the `@`-family words.

`DS@` S QFT

Print the Data Stack content as unsigned numbers in a non-destructive way. I built it to check the stack state after a system freeze caused by wrong functions during the implementation phase; since this is a super-immediate word, it needs no dictionary search and so it's likely to work when `.S` cannot. And since it prints unsigned numbers, it could reveal strange unwanted numbers, because compilation codes are always positive.

GETSTATE S QFT

Show the current STATE on the screen, followed by a Carriage Return. It prints 0 or 1, with the meaning that

```
0 = INTERPRET STATE and
1 = COMPILE STATE
```

This was done because, sometimes, when the system was under construction, I needed a fast way to check the STATE, without interfering with the running system and the stacks.

PREVHERE -- addr S QFT

Leave the address of the previous compiled word. Differently from LAST, which returns the address of the last word under compilation, regardless of the compilation success, PREVHERE returns always the last **successful** compiled address, and it's the same value used internally by qartus to traverse the whole dictionary. It was built when testing the word creation process, to dump memory in a fast way.

RS@ S QFT

Print the Return Stack content in a non-destructive way. In normal usage, it should always show an empty Return stack. I built it to check the values left by the built-in DO..LOOP family words, or to check scattered values left behind.

SH S QFT

Used in the form:

```
SH <name> <text>
```

to invoke the command <name> (which is any tool in the path or a shell command) and passes it to the shell, along with all arguments <text>.

SH arguments extend until the end of the input buffer, so that it must not be followed by any other Forth word (that will be interpreted as an argument).

Remember also that while SH can be written in lower or upper case, the rest of the command is passed as-is, because the shell is case-sensitive.

SYS S QFT

Start a shell session. The shell is the one defined in `custom.h` and by default it is set to `/bin/bash`. The shell starts from the current directory; during the shell session, the directory can be changed but when the shell command `exit` is typed, the qartus session is restored in the original directory. At all effects, apart the intervening shell session, nothing changes in the qartus session.[90] This command may also be typed as MON (from a fig-Forth tradition).

Technical speech: the shell is started using the C function `system()`, which creates a fork in the executable, and the created fork starts the execution of a shell session. As such, the shell does inherit the last saved shell state. If you have started qartus from the shell, and after a while you type `SYS`, the current shell history will not contain the commands typed so far, because the hosting shell, which is underneath qartus, has not rewritten the history file yet.

3.7. The integration file qlibrary.ft

The package includes the file `qlibrary.ft`, a library file that contains a great number of useful words definitions in Forth (for instance `INDEX`, `S"`, `CLEARSTACK`); some of them fill some holes in the Forth-79 word set (for instance `<>`, `><`, `m*`) and some are ANSI 1994 words in a Forth-79 translation (not always fully respondent to the ANSI 1994 Standard). Some are Forth-83 renditions, but be careful because the latter overwrite the Forth-79 versions.

The library file is copied in the user `$HOME/.qartus/` directory by the procedure --config (see the file INSTALL).

You can safely load the entire library with `INCLUDE` (the qlibrary protects itself from the Forth-83 versions), or you can load only the words you need with `ENSURE` or `REQUIRE`. See `INCLUDE`, `ENSURE` and `REQUIRE`.

All library words, being written in Forth, can be fully disassembled with `SEE`.

If you have built your own library, you can use it as the current source by setting the library reference with `LIB`. Remember that, to work smoothly, libraries should only have colon words definition or variables/constants instantiation (and all possible comments), but not directly executable code. See also `LIB`.

[90] Why not setting qartus as a shell environment? Add it to the shell list (adding it to `/etc/shells` as root) and have fun with qartus! For all shell needs, type `SYS` and you're done! You can also start your session with `$ qartus --init=sys` and start a shell session *before* Forth!

4. The built-in debugger

qartus has a built-in debugger engine, a simple words-tracking with stack content picturing, enabled with the words `ON DEBUG` and `OFF DEBUG` to enable/disable the debug on the whole program and with the word `TRACE` to enable/disable the debug on a single user word.

Don't expect a very complex debugger. Its aim is to signal where a word fails by printing the flow of procedures involved and the stack content at each step.

4.1. Using `DEBUG`

Suppose for instance that we have a word defined as such:

```
: DE-TEST  ( d1 n1 -- n2 )
   ROT ROT SWAP /
   SWAP / ROT
;
```

(it's a useless word, just to show how the debugger works). Now let's try it:

```
56783344. 567 DE-TEST
? empty stack.
56783344. 567 DE-TEST
       ^
```

What we know so far is that `DE-TEST` fails. But where? We have no other suggestions by the standard error detection system.

The debugger, when enabled, at each step shows us some more information, namely the Program Counter, the Data Stack in signed mode, along with the name of the word or the number under evaluation; the following shows the entire debug session:

```
ON DEBUG   ok
56783344. 567 DE-TEST
<3> 29168  866  567

PC:03639  Executing built-in word --> rot <--

<3> 866  567  29168

PC:03641  Executing built-in word --> rot <--

<3> 567  29168  866

PC:03643  Executing built-in word --> swap <--

<3> 567  866  29168

PC:03645  Executing built-in word --> / <--

<2> 567  0

PC:03647  Executing built-in word --> swap <--

<2> 0  567
```

```
PC:03649  Executing built-in word --> / <--

<1> 0

PC:03651  Executing built-in word --> rot <--

? empty stack.
56783344. 567 DE-TEST
              ^
```

The problem is on the third ROT command, which finds less than three items on the stack, and thus rightly complains. Of course, at this point, it's up to us programmers review the algorithm and find a solution for this, but at least we now know where the problem lies. Sometimes, this is just the thing to know, isn't it?

4.2. Using TRACE

If you want to select a specific user word for debugging, you can use TRACE; this word will trigger debugging only on the word whose name has been given as argument; if, for instance, the previous word DE-TEST must be inside another word like this:

```
: WHAT    ( -- )
   GET-VALUE DUP IF DE-TEST THEN RESULT
;
```

enabling debugging with DEBUG and executing WHAT would print the debugging information for all the user words WHAT, DUP, GET-VALUE, DE-TEST and RESULT; if instead you invoke:

```
TRACE DE-TEST
```

before executing WHAT, the debug will be active only during the execution of DE-TEST; this lets inspecting the program to find more exactly where it fails, without the need to traverse all words, ending in a long debugging output, where the useful information must be searched.

If the TRACEd word has some user words in its body, these words are not TRACEd (that is the debugger does not 'enter' into these) so that, in the assumption these inner words work regularly, you can concentrate on the TRACEd word (DE-TEST in the example) without bothering for the rest.

TRACE and DEBUG mutually exclude themselves, because they interfere with each other operations. So use one or the other, on the need.

A final remark: TRACE cannot be set on built-in words (variables included); moreover, as you can easily see by yourself, tracing user variables or constant is useless, because they don't perform any operation. I must also observe here that CREATEd words always perform the operation of returning their PFA, and this pushing is shown by the debugger; all DOES> operations (if any) are shown afterward.

4.3. Debug Commands

At each step, the debug waits for one keyboard key (lowercase or uppercase are the same, no ENTER key):

C - Continue in debug mode
if the key pressed is 'C', the program continues in debug mode (the tracepoint is not removed), but without steps interruptions, showing the debug lines until the natural end.

D - Data stack

if the key pressed is 'D', the Data Stack is printed (in the same style of `.S`) with unsigned values. Useful for inspection of addresses in the stack. The stack is identified with `DU:` (for Data Unsigned).

`E` - Execute user code
if the key pressed is 'E', the system suspends the current word and requests a line of text (the string `(suspended)` appears, the prompt becomes `(D)`); the input text is evaluated by the interpreter as Forth code (by calling the line interpreter in debug-off mode); this feature can be used to inspect variables content, modify the current stack or the system state, `DUMP` memory or change the program behaviour during runtime, or any other legal Forth operation[91]. After executing this user code, the string `(resumed)` appears, the debugger resumes control, restoring the system parameters and the previous input line, waiting for a key press to execute the suspended word or another debug command.

`F` - Control flow stack
if the key pressed is 'F', the Control Flow Stack is printed (in the same style of `.S`) with signed values. The stack is identified with `CF:` (for Control Flow).

`G` - Exit debug and go on
if the key pressed is 'G', the tracepoint is removed, the `DEBUG` flag is disabled and the program continues in normal debug-off mode, until its natural end. The debug is reset, so if you want to debug again, re-enable it with `TRACE` or `ON DEBUG`.

`H` - Help
if the key pressed is 'H', a brief help is shown with the titles of this help. Refer to this manual for more info.

`R` - Return stack
if the pressed key is 'R', the Return Stack is printed (in the same style of `.S`) with unsigned values; in case of `DO..LOOP` indices, the negative values are shown as unsigned, so be aware of this. The stack is identified with `RU:` (for Return Unsigned).

`X` - Stop and exit debug
if the key pressed is 'X', the program and the debugger stop; the tracepoint is not erased and the `DEBUG` flag is maintained, for a subsequent debug session. The string `(terminated)` appears on the screen.

Any other key gets to the next step.

[91] Debugging uses its own circuit to get the input commands; therefore, if you invoke `BYE` as the user code, the system won't exit until the next iteration resumes. Using `COLD` as the user code is legal, but since debugging or tracing are not disabled, they continue after the dictionary reconstruction, with unpredictable effects (but they usually stop regularly because an empty cell is found).

5. Differences with the Forth-79 Standard

Here are collected all the differences of qartus with the Forth-79 Standard, split in two categories: the changes (that modify the Standard behaviour) and the enhancements (that increase the Standard limits). Let's begin.

5.1. Changes

5.1.1. Word sets

qartus has the following word sets hardwired as built-in:

- The DOUBLE word set (defined in the Forth-79 standard as Double Number word set in the Extension Word Sets chapter - page 32) is implemented by default and belongs to `FORTH`'s vocabulary.
- The `EDITOR` vocabulary is the only other system vocabulary added, at present.
- The ASSEMBLER word set (defined in the Forth-79 standard as Assembler word set in the Extension Word Sets chapter - page 34) is currently not present.

5.1.2. `-->`

This word does not belong to the core set of the Forth-79 Standard; it belongs to the Reference Word Set, where it's stated that it "*May be used within a colon definition that crosses a block boundary*" (i.e. the colon is on one block, the semicolon on the next); this is not true for qartus: any definition must be accomplished into the block, and thus the effect of `-->` is merely 'append' the next block.
Moreover the Standard does not specify what should happen in case the block does not exist or is not found. I chose to ignore the error condition, after the returning of a block filled with blanks by `BLOCK`, which means the compilation will simply stop, and this may mean another difference, if not with the Standard, with many Forth-79 compilers.

5.1.3. dot-quote `."`

The `."` word in qartus behaves a little differently if the input stream is exhausted before the terminating double quote; while the Standard asserts that "*If the input stream is exhausted before the terminating double-quote, an error condition exists*", qartus is more tolerant, and the string is regularly and implicitly closed.

5.1.4. `BASE`

The Forth-79 Standard specifies the limit of {2..70} for the available bases for numbers. qartus, in a more modern feeling (and in the assumption that this is enough) limits this range to {2..36}, adhering thus to the modern languages range, proved to be adequate and robust.

5.1.5. `BLOCK` and `BUFFER`

The Forth-79 Standard specifies for `BLOCK` that "*...if correct mass storage read or write is not possible, an error condition exists*", without specifying if this error condition is originated by the absence of the file relative to that block, or by a disk error or what. As a result, I built `BLOCK` to return a full empty block (filled with blanks) if the relative file can't be loaded (because it does not exist or because you haven't the access rights); the problem reported in the Standard arises when the *saving* of the block is not possible: whence the error condition. Besides, in qartus `BLOCK` and `BUFFER` use an unsigned argument, to let the user treat up to 65535 block files (excluding zero).

5.1.6. CONTEXT

The definition of CONTEXT in the Standard is: "*Leave the address of a variable specifying the vocabulary in which dictionary searches are to be made, during interpretation of the input stream.*". fig-Forth defined both CONTEXT and CURRENT to be pointers to the vocabulary word within which dictionary search will first begin, and this, someway, has set the behaviour of the following Forth-79 compilers. In qartus, CONTEXT returns an address where a control-bit byte is stored, specifying a given vocabularies tree; this value is checked, in run-time, against a given word's definition, to see if it matches the dictionary control-bit value, in which case it is executed. This is a feature, more than a difference with the Standard, but this prevents the usage of, for instance, `CONTEXT @ @` to return the reference of the context vocabulary, whereas in qartus it suffices writing `CONTEXT @`.

5.1.7. CURRENT

The definition of CURRENT in the Standard is: "*Leave the address of a variable specifying the vocabulary into which new word definitions are to be entered.*". In qartus, CURRENT returns an address where there's a control-bit byte specifying a given vocabularies tree; this value is put into the definition being created in the dictionary control-bit, value that will be checked in run-time to see if it responds to the CONTEXT control-bit value, in which case the word is executed.

5.1.8. DPL

In qartus, DPL is affected by the user input and also when a double number (with dot) is pushed to stack, *á-la* fig-Forth. This is a violation of the definition of DPL in the Reference Word Section, where it's stated "*...DPL may be set explicitly, or by certain output words, but is unaffected by number input*".

5.1.9. FORGET

In qartus, FORGET does not complain if the word to be forgotten is not found, doing nothing in return. The error instead is explicitly ruled in the Standard, where it's stated "*Failure to find <name> in CURRENT or FORTH is an error condition*". However, the Standard imposed error condition prevents its usage as an automatic tool. See the note for the FORGET entry in this manual.

5.1.10. LITERAL and DLITERAL

The Standard specifies for LITERAL the compilation behaviour without specifying any execution behaviour, but also without setting LITERAL as compile-only (a feature later enabled in Forth-83). So I turned to the Forth-83 scheme, which seems smarter, and enabled LITERAL as compile only, with the consideration that typing from console, e.g., `24 LITERAL`, is meaningless. The same for DLITERAL.

5.1.11. PAGE

In the Reference Word Set (page 8) , PAGE is set to "*clear the terminal screen or perform an action suitable to the output device currently active*". In qartus PAGE acts always on the terminal screen.

5.1.12. WORD

The Standard requires WORD to ignore leading occurrences of the terminator character. qartus, instead, ignore the terminator only when it's a blank, letting treating the empty string case. See the WORD entry in the dictionary.

5.2. Enhancements

5.2.1. Memory

The Standard requires memory to be organized in single bytes (i.e. 8 bits) for characters, in one *cell* (i.e. two bytes, or 16 bits) for single numbers and two *cells* (i.e. four bytes, or 32 bits) for double numbers.

qartus uses the same number structure, but uses 32-bit numbers (corresponding to the C `int` type) for the basic memory **unit** (the Forth-79 `byte`), replicating the Forth-79 memory model (i.e. one single *unit* for characters, two *units* for 16-bit numbers and four *units* for the double numbers), using for each unit only the lower 8 bits[92] (thus, a *unit* corresponds to a `byte`). The full 32-bit unit capacity is reserved for inner addresses and specific markers used in compilation.

The starting address of a word definition in the dictionary is not a 'cell' (two units) but a full 32-bit number. This is done for speed reasons. The interpreter (and mainly `EXECUTE`and all the built-in words) skips the bytes before the Compilation address (a fixed width) and knows what to do within each instruction. If you want to read this cell address, `@` and `2@` are of no use. qartus's programmer can scan the memory with `DUMP`, and interact with the special 32-bit numbers by means of the elective words `32@ 32,` and `32!`, with double numbers arguments, for storing and reading this extra-bit values.

5.2.2. The `(` comment

The `(` comment has a feature that the Standard does not require: if the closing parenthesis `)` is not followed by a space, a TAB or by the end-of-line, it is not considered as a closing parenthesis, and the comment continues. See the note for the `(` entry in this manual.

5.2.3. `CREATE`

In qartus, `CREATE` does not simply create a dictionary entry. Rather, it reserves a coded sequence that is used by `FIND` and `'` (tick) to distinguish the CFA from the PFA of a `CREATE`d word.

This is not really a difference, it's more like an implementation feature, but in any case, here it is.

5.2.4. `DOES>`

The `DOES>` instructions inserted in a defining word are inherited by the defined word, and this is the normal usage of `CREATE..DOES>`; but qartus has a property that the Standard does not specify or clarify: if the defining word with a `DOES>` section is used as a defining word itself in another defining word, and this second defining word is used to define a word without `DOES>`, this last word inherits the `DOES>` section of the original defining word; an example is thousand words:

```
: ARRAY   CREATE   DOES>  2+ @ ;
: NEWARRAY   ARRAY ;
NEWARRAY MYLIST
```

In this example, `ARRAY` is the original defining word with a `DOES>` section, `NEWARRAY` is a defining word without a `DOES>` part, but defined through `ARRAY`; `MYLIST` is defined by means of `NEWARRAY`, and inherits the `ARRAY DOES>` section, because `NEWARRAY` has none.

[92] I know that this may result, at a first glance, in a great waste of memory, but in a 64-bit Operating System (and maybe more significantly in a 32-bit one) the execution of 32-bit numbers is faster than using the `short` or `char` C-types.

5.2.5. EMIT

In qartus, EMIT uses the full {0..255} range for output, while the standard specified the {0..127} range.

5.2.6. TYPE

In qartus, when TYPE is used to print n characters from a specific address and if a zero occurs in the characters queue, it stops printing the rest, even if not all n characters have been output and the rest is not entirely null. In the standard, there's no mention about this feature, but since all strings are terminated by zero, I guess all Forth-79 interpreters did the same, because printing the rest of the string could ruin the output with random characters.

5.2.7. WORD

qartus dedicates to WORD a fixed and independent buffer, 1024-characters long. In the Standard it's not clear whether this should be independent and fixed, or variable in dimensions and positions (like PAD, for instance), even in *Forth Dimensions* Vol. III n. 5, Robert L. Smith discusses the topic asserting that placing the WORD string at HERE is non-standard (it was a fig-Forth feature). In any case, this feature lets the programmer use WORD safely because its memory space does not belong to dictionary memory.

6. Forth-83 and beyond (how evolution has won)

The release of Forth-79 established a firm point in the Forth development. But Forth scientists, in the following years, posed a number of questions that the Forth community couldn't ultimately ignore any more:

For instance, John Arkley, in "*Forth Dimensions*" Vol. IV, N. 3 September/October 1982, writes: "*79-Standard FORTH fails to address the problems of directories, directory management, file creation, device definition, device assignment, special driver linkages, keyboard mapping, function keys, and the like. The philosophical reason is that no one operating system will ever do what is desired, and it usually hinders rather than supports the programming language system*". Or read what Robert L. Smith writes in Vol. IV, N. 2 July/August 1982: "*The area of greatest concern for the next Standard is that of vocabularies. Forth-79 has a very weak vocabulary structure. It was chosen as the minimum subset of most FORTH implementations*".

All these instances led to the Release of Forth-83 in early 1984. In the following, many concepts of Forth-83 are discussed; keep them in mind if you want to translate Forth-83 code for qartus (which is a 79-er). Keep also in mind that the ANSI 1994 is a direct evolution of Forth-83, and that many concepts apply also to it.

6.1. The Compilation Address

In Forth-79, four addresses are used to name different fields in a word definition in the dictionary (the fig-Forth was the first to use these definitions): the Name Field Address (NFA), the Link Field Address (LFA), the Code Field Address (CFA), and the Parameter Field Address (PFA). To enhance code portability, Forth-83 uses only one address, the Compilation Address (roughly equivalent to the Code Field Address). The Compilation Address is the one now returned by `'` and `FIND`, compiled to colon definitions, and used by `EXECUTE` to run the word definition. Only one extra word is provided in the Forth-83 Standard to access information stored in the parameter field: `>BODY`.

6.2. To be or `NOT` to be

In Forth-83 the operator `NOT` was built according to this definition:

```
NOT          16b1 -- 16b2                    83
16b2 is the one's complement of 16b1.
```

This differed from the definition in Forth-79 which was:

```
NOT          flag1 -- flag2                165
Reverse the boolean value of flag1. This is identical to 0=.
```

The difference is that, while the one's complement is performed by inverting all the bits in a number (and this justified the turning of the True value to -1, because by inverting all bits of zero, one gets -1), the inversion of the boolean value means turning zero to a not-zero value or turning any not-zero value to zero, without taking into account the bits state.[93]

6.3. `FIND` and `'` (tick)

In Forth-83 `FIND` operates differently than in Forth-79, and expects the string to be searched as a counted string address, rather than searching it in the Input Buffer. The `'` (tick) instead returns the Compilation address, rather than the Parameter Field Address.

[93] Read, for instance, the article "*Representation for Logical True*", by Robert L. Smith, in "*Forth Dimensions*", Vol. IV, N. 4 November/December 1982.

6.4. Unsigned index arguments for memory commands

The words BLANK, FILL, MOVE and MOVE> in Forth-83 use unsigned index values.

6.5. Removal of STATE-dependent words

Many Forth-79 words work both in the execution mode or compiling mode, with different behaviours, like LITERAL, ' (tick), ." (dot-quotes), for instance. In Forth-83, the STATE-dependent words are either eliminated or separated in interpretation-only or compilation-only words, making the system less ambiguous. For instance, ' is split into two words: ' (interpreted version) and ['] (compiled version), and ." is split into .((interpreted version) and ." (compiled version).

6.6. Improved DO..LOOP

The major change in DO..LOOP was that in Forth-83 LEAVE is made to terminate the loop immediately upon its execution rather than setting the index of the DO..LOOP to the terminating value.

6.7. Division type

Division in Forth-83 is floored towards negative infinity, instead of rounded towards zero. The quotient and modulus have now a smooth change between positive integer and negative integer domain when either divisor and/or dividend is negative. This lead to the change of the definitions for /MOD and */MOD, which now return a remainder equal in sign to the divisor, rather than to the dividend.

6.8. Zero-based PICK and ROLL

In Forth-79, ROLL and PICK were 1-based, that is the stack count numbered the top item as 1. In Forth-83 the top of the data stack is treated as a memory area: addressing into this area is zero-based like addressing regular memory areas.

6.9. Improved WORD

WORD was improved making it return a blank after the returned string, while the Forth-79 explicitly declares that the current delimiter found is stored at the end of the definition and excluded by the count.

6.10. Renaming of some words

The following words were renamed, with substantially identical properties:

- U/MOD was renamed UM/MOD
- U* was renamed UM*
- <CMOVE (that was in the Reference Word Set) was renamed MOVE>
- BLANKS (that was in the Reference Word Set) was renamed BLANK
- 79-STANDARD was (of course) renamed FORTH-83.

7. Conversion of fig-Forth programs

I'm sure you all have a lot more experience than me in finding differences between Forth-79 and fig-Forth and knowing how to convert a program from one environment to another.

So, I present here only what I found in my experience while writing programs for qartus, having to decide whether the faults in the program were due to errors in my implementation or to a precise difference in behavior with respect to the original Forth environment from which I was translating the software.

fig-Forth is, for Forth-79, the closest cousin, quite a twin actually, for many reasons. They were conceived in the same period (late Seventies), and the Forth-79 Standard was built using the fig-Forth experience.

But not all was plainly passed to the Standard. Some important features changed, making the two environments not completely compatible.

Here is what I found.

7.1. Can I say a `WORD`?

The word `WORD` (you'll excuse the forced pun) has passed a robust change in passing from fig-Forth to Forth-79; here are the two definitions cores:

fig-Forth:
`WORD  c  ---`
"[...stores...] *the packed character string beginning at the dictionary buffer* `HERE` [...]"

Forth-79:
`WORD  char -- addr`
"*The characters are stored as a packed string with the character count in the first character position* [...]. *The address of the beginning of this packed string is left on the stack.*".

The difference is subtle but important: Forth-79 does not involve `HERE` anymore. The address may be any (a reserved dictionary space, a dedicated buffer, or `HERE` again), and it's left on the stack (incidentally, this explains why in fig-Forth it's frequent seeing the token `WORD HERE`). In qartus, by the way, `WORD` has a dedicated buffer.

To convert this word from fig-Forth, the word `HERE` must be omitted (because the address of the counted string is already on top on stack after the execution of `WORD`).

The evolution of Forth is not favorable to `WORD`: it will receive more critics after the release of the ANSI 1994, because of the bad features here described, in favour of the new ANSI definitions `PARSE` and `PARSE-NAME` (see for instance gforth). `WORD` will remain in the Standard (up to now, at least) only for back-compatibility issues. A bad destiny for such a historical word...

7.2. `+LOOP` limits

The two definitions in the two manuals are quite the same:

fig-Forth:
"*..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).*"

Forth-79:
"*...Return execution to the corresponding DO until the new index is equal to or greater than the limit (n>0), or until the new index is less than the limit (n<0).*"

The real difference is when +LOOP is used with negative increments; while Forth-79 includes the limit, fig-Forth explicitly avoids it. So if something like

```
0 10 DO ... -1 +LOOP
```

is written in fig-Forth, the Forth-79 equivalent instruction is

```
1 10 DO ... -1 +LOOP
```

7.3. Using VARIABLE properly

The fig-Forth VARIABLE needed an initialization parameter on the stack:

```
10 VARIABLE ICONS
```

Of course, the Forth-79 version of VARIABLE is slightly different, and the previous line must be translated as:

```
VARIABLE ICONS  10 ICONS !
```

Sometimes, you may find a declaration like

```
0 VARIABLE NAME 28 ALLOT
```

This is writable in Forth-79 as a plain translation:

```
VARIABLE NAME 28 ALLOT
```

with the same effect on program, but the proper way to do it in Forth-79 is

```
CREATE NAME 30 ALLOT
```

which offers a visual declaration of the dedicated space. The same must be said for 2VARIABLE (considering that 2VARIABLE allocates four bytes at start).

7.4. A SIGN of destiny

Curiously (and unexpectedly) the word SIGN was changed passing from fig-Forth to Forth-79; the change is small and, since then, it has been maintained in Forth83, ANSI 1994 Forth and probably will also be in next Standard. The fig-Forth stack picture is

```
SIGN      n d -- d
```

while the Forth-79 stack picture is

```
SIGN      n --
```

Since it is always used after #S, which leaves a double number on the stack (a double zero), the global effect of SIGN in fig-Forth is to perform an internal ROT, to surface the signed value n to evaluate; the fig-Forth usual figure is:

```
<# ..... #S SIGN #>
```

See for instance the Alan Winfield book *The Complete Forth*, and precisely the example of .$ at par. 8.4,

where he uses `SIGN` directly after `#S` (he claims to use Forth-79 but his R-Forth compiler has some strange relations with fig-Forth...)

The Forth-79 `SIGN`, instead, looks for the signed value on the top of stack (more naturally, I must say), so the `ROT` must be explicitly performed by the programmer; here is the Forth-79 usual figure:

```
<# ..... #S ROT SIGN #>
```

See for instance the *Forth Programmer's Handbook* page 91, where the `.` (dot) and `NNN` words are designed, both using `ROT SIGN`.

7.5. How to `FIND` it?

The word `FIND` of Forth-79 passed a robust change passing from fig-Forth to Forth-79. Here are the two definitions:

fig-Forth:
"`-FIND --- pfa b tf (found) --- ff (not found)`
Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left".

Forth-79:
"`FIND  -- addr`
Leave the compilation address of the next word name, which is accepted from the input stream. If that word cannot be found in the dictionary after a search of CONTEXT and FORTH leave zero".

Changes are many:

* the name was changed from `-FIND` to `FIND`
* the return value was changed from the parameter field address to the compilation address
* the length byte and the true flag were removed

To use a version compatible with `-FIND` you can `ENSURE` it from the library (the word simply finds the compilation address or zero, and if not zero converts it to PFA and then adds the length byte and the true flag).

7.6. A `VOCABULARY` lookup

The `VOCABULARY` word, in my opinion, was better defined in fig-Forth than in the Forth-79 Standard; let's look at the definitions:

fig-Forth:
"*A defining word used in the form:*
`VOCABULARY cccc`
to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.
In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK".

Forth-79:

"*A defining word executed in the form:*

```
VOCABULARY  <name>
```

to create (in the CURRENT vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent execution of <name> will make it the CONTEXT vocabulary. When <name> becomes the CURRENT vocabulary (see DEFINITIONS), new definitions will be created in that list.
In lieu of any further specification, new vocabularies 'chain' to FORTH. That is, when a dictionary search through a vocabulary is exhausted, FORTH will be searched".

The difference is not subtle at all. fig-Forth **does** specify that all vocabularies are linked according to their creation, and ultimately link to `FORTH`, while Forth-79 simply attests that vocabularies are simply chained to `FORTH`, leaving some speculation behind.

But any possible speculation (see the chapter "VOCABULARY problems", for instance) lead to one conclusion only: no Forth-79 compiler must differ from the fig-Forth behaviour. My humble opinion is that the Standard fails in clearly defining the matter.

7.7. Where is it?

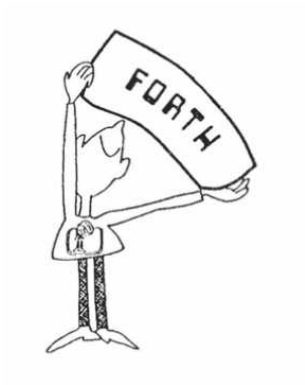
The introduction of the 79 Standard, after many years in which fig-Forth was the *de facto* Standard, deliberately changed the name of some words that was current cash in fig-Forth, without really changing the run-time behaviour, but causing some mental displacements in old fig-Forth users. I suppose the conversion of fig-Forth programs to Forth-79 revealed quite painful, in those first years.

Anyway, here is the list that I know of commands that work quite exactly like the Standard, but their name is not the Standard one:

fig-Forth	Forth-79
-DUP	?DUP
<BUILDS	CREATE
DMINUS	DNEGATE
IN	>IN
MINUS	NEGATE
R	R@
U/	U/MOD

8. My favourite FD Articles

I list and comment here some of the *Forth Dimensions* articles that I found most interesting, especially those related to Forth enhancement, algorithms, games, math and its derivatives. I know that many of you, perhaps everyone, know them better than me, but I hope that your interest in these articles is at least one-tenth of mine; that you, as me, can appreciate them, as flashes of intelligence woven with veils of time.



(Note: reproduced without permission. If its authors should ask for removal, this image will disappear in the next version.)

The numbers in square brackets refer to the blocks (included in the package) where the software described in the article (translated to qartus and anyway tested) is written, using, as accurately as I could, the original text output; use `LOAD` to load the screens with the first block number on top of stack. If a name is reported in square brackets, it refers to the file sample (also included in the package), to be loaded from console as an argument of qartus (in this case, read the source to know how to use the program); use `INCLUDE` to load the file from within qartus's interface.

Extensibility with Forth, by Kim Harris
Vol. I, N. 2 August/September 1978
A quasi-philosophical lesson about the power of Forth in solving any kind of problems.

Threaded Code, by John James
Vol. I, N. 2 August/September 1978
A brief essay about Forth's memory. After reading this article I thought: "Why not writing a Forth compiler by myself?"[94]

Forth-85 "CASE" Statement, by Richard B. Main
Vol. I, N. 5 January/February 1980
Interesting study, the first appeared in this review with this topic, aimed to implement `CASE` (in a different way than the one now standard).

Forth, The Last Ten Years and The Next Two Weeks..., by Charles Moore
Vol. I, N. 6 March/April 1980
Ladies and Gentlemen: Charles Moore! 16 pages of history of Forth, experiences, a bit of projects, by the Forth's creator... This is the audiotape transcripts of the Forth Convention held in San Francisco in October 1979, only months before the Forth-79 Standard issue. The fig-Forth member Jim Berkey recorded the technical sessions and the banquet speech by Charles Moore, along with the questions posed to Moore and the following answers.

Forth in Literature, by W.F. Ragsdale

[94] Now I can tell you that this experience was like entering Doom!

Vol. II, N. 1 May/June 1980 **[20108-20110]**
An automated speech generator: The Theory that Jack Built.

Recursion - The Eight Queens Problem, by Jerry LeVan
Vol. II, N. 1 May/June 1980 **[20057-20059]**
A fast algorithm to give all solutions to the Eight Queens problem.

Temporal Aspects of the Forth language, by John M. Derick
Vol. II, N. 2 July/August 1980
A very instructive article about Forth, directed to programmers with different experiences not belonging to the Forth world.

Towers of Hanoi, by Peter Midnight
Vol. II, N. 2 July/August 1980 **[20012-20017]**
The algorithm for the Towers of Hanoi, solved with a direct translation from a Pascal version. I had to introduce some more delay (namely increasing the `DELAY` parameters and adding a call to `DELAY` in `TRAVERSE`) because computers nowadays are too fast![95] To stop the execution, press any key.[96]

Random number generator, by Peter H. Helmers
Vol. II, N. 2 July/August 1980 **[20044]**
A simple but useful random number generator algorithm, which appears on the last page of the issue.

CASE contest (monography)
Vol. II, N. 3 September/October 1980
The whole number is devoted to the `CASE` structure proposals, submitted by many, who show their own point of view. The first (titled "*Just in CASE*", by Charles E. Eaker) is the only one exactly equal to the current Standard version. So I assume (I don't know it for certain) that Mr. Eaker is the `CASE` winner!

The execution - Variable and array, by Michael A. McCourt
Vol. II, N. 4 November/December 1980 **[20502] [xeq.ft]**
This article shows the power of Forth in dealing with deferred execution. Unfortunately, the code for `()XEQ` (deferred execution in arrays) is quite unreadable; so I limited the transcription to the `XEQ`, `INSTALL` and `IN` words only. If you can do better than me, let me know.

The CASE, SEL and COND structures, by Peter H. Helmers
Vol. II, N. 4 November/December 1980
This article shows some interesting structures for multiple selection (all predating the `CASE` structure of Forth-83); these structures, adopted from URTH, the Forth system at the Rochester University, didn't turn to be widely used, but it's nonetheless an interesting point of view, in times when the language was still magmatic.

Programming Aids & Utilities, by Kim Harris
Vol. III, N. 1 January/February 1981
A collection of utilities screens in Forth. Many of them are so common that I put them in `qlibrary.ft`.

Product Review: Timin-Forth", by C.H. Ting
Vol. III, N. 1 January/February 1981 **[*bench.ft]**
In this small but interesting article, Mr. Ting presents the Timin-Forth for CP/M, by Mitchel[97] E. Timin. He then compares this CP/M Forth with the Z80 Forth by Ray Duncan using six benchmark programs.

[95] The original algorithm showed no progresses and I thought it was a failure in qartusl, or maybe the algorithm was bad conceived, or maybe there was a error of trascription; figure out how many time I spent trying to understand that there were no failures but only that the program was too fast!

[96] The program is executed in fig-Forth mode, and it is not reset to default at the stop. So be sure to set `FIG-STATE` to zero if you want to restore the Forth-79 mode afterwards.

[97] Sic.

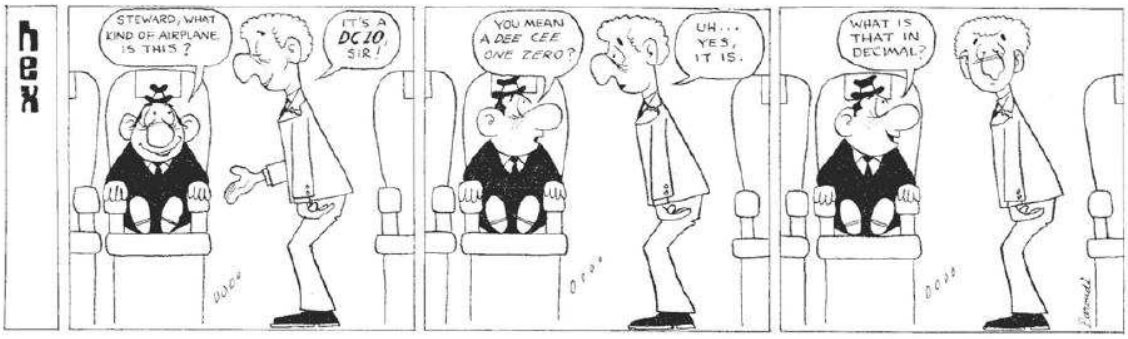
I was so intrigued by this test that I built a bench test for qartus using four out of the six words (the remaining are less interesting), and translated the code for gforth and swiftforth. I know it's quite unfair, but here are the comparison results of the original behaviours with the timing of the same program ran through my current computer (a laptop with 4 processors Intel Core i3-5005U @ 2.00 GHz, arch. x86_64) adapted for qartus, gforth and Swiftforth:

```
Absolute values (seconds):
              Z-80     Timin   qartus    gforth      Swiftforth
-------------------------------------------------------------
test 1 (loop)  2.9      2.3    0.00129   0.000143     0.000010
test 2 (-)     7.4      5.9    0.00490   0.000233     0.000010
test 3 (*)    54.9     44.0    0.00469   0.000239     0.000010
test 4 (/)    88.6     74.3    0.00440   0.000231     0.000010

Speed ratios (base=Z-80, the slowest):
              Z-80     Timin   qartus    gforth      swiftforth
-------------------------------------------------------------
test 1 (loop) 1.00     1.26  2,248,06    20,279.7     290,000.0
test 2 (-)    1.00     1.25  1,510.20    31,759.6     740,000.0
test 3 (*)    1.00     1.25 11,705.76   229,707.1   5,490,000.0
test 4 (/)    1.00     1.19 20,136,36   383,549.8   8,860,000.0
```

qartus is (35 years after) at least 20000 times faster than the Z80 Forth in 1981 (for test 4, division). I'm not so proud of my results, though: what about gforth? 380000 times faster (for the same test 4)! And what about the astounding and astonishing result of Swiftforth, which is (for the very same test 4) 8.8 million times faster? I still have a long way to go...
You can find the benchmarks files in the `files/` directory of the installation folder: `ybench.ft` (for qartus), `gbench.ft` (for gforth) and `sbench.ft` (for Swiftforth).

HEX strip, by Baroudi
Vol. III, N. 1 January/February 1981
A humorous strip, understandable only by computer scientists like you...

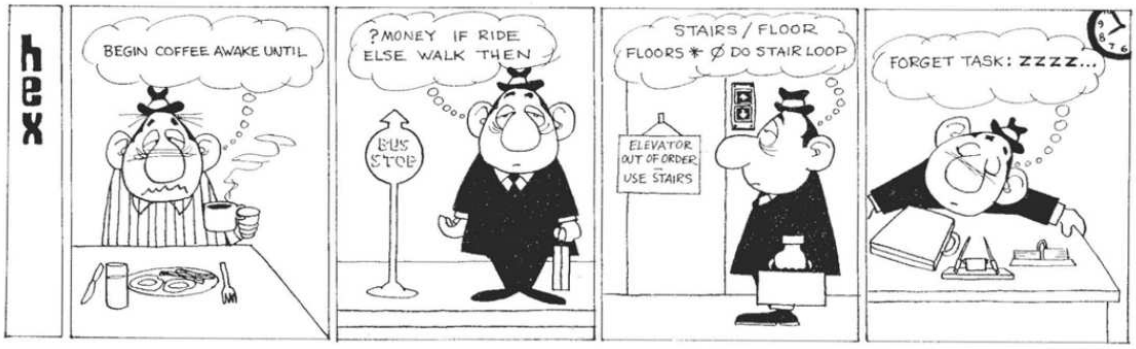


(Note: reproduced without permission. If its authors should ask for removal, this strip will disappear in the next version.)

Compiler Security, by George W. Shaw III
Vol. III, N. 1 January/February 1981
Some notes (in a long article) about the security issue on fig-Forth. The article is too much fig-Forth-oriented to be fully compatible with a Forth-79 compiler, but it explains in detail how fig-Forth error-dealing words can help in validating Forth code in many fields: structural errors, parameters errors, machine state errors; The conclusion is: "*Should we have all the protection all the time, or just some of it and a programmer protection package? Or maybe there is a better alternative.*".

Userstack, by Peter H. Helmers
Vol. III, N. 1 January/February 1981
A proposal for a User Stack to avoid the problems involved with the Return Stack. Unfortunately, this interesting article cannot be converted to qartus because it uses Machine Language targeted for the 8080 CPU. Holy patience...

HEX strip, by Baroudi
Vol. III, N. 2 July/August 1981
Another humorous strip, this time understandable only by Forth programmers like you...



(Note: reproduced without permission. If its authors should ask for removal, this strip will disappear in the next version.)

Forth Standard Corner, by Robert L. Smith
Vol. III, N. 3 September/October 1981
This article, which is about the `WORD` implications, had such a strong impact on me that I modeled the `WORD` definition in this manual following some of Mr. Smith suggestions.

The Forth-79 Standard - A Tool Box?, by George W. Shaw
Vol. III, N. 3 September/October 1981
A very interesting but small article about the definition of transportability of Forth-79 environments (which translates to me in "*how far can you go in adding new not-compatible statements to your compiler?*"). Here is the conclusion: "*When considering the Forth-79 Standard, treat it as a basic tool box. Additional tools are applications from the point of view of the standard. Extend it as necessary. Can you add what you need by defining it only in terms of standard words? If not, what is the minimum necessary to allow you to do that: more definitions or more explanations? Experience is all that will tell. Send in your results.*".

Book Review: Starting Forth, review by George W. Shaw
Vol. III, N. 3 September/October 1981
This is not a book review. This is THE book review. About the book of the books about Forth. About the author of the authors about Forth. The Master. The Teacher.

Recursion and the Ackermann Function, by Joel V. Petersen
Vol. III, N. 3 September/October 1981 **[ack.ft]**
A recursion study which is totally compatible with qartus; the computation limits are the limits for the data and return stacks (1024 cells); I don't know which were the stacks limits for the NIC-Forth used in the test, but the study results for the NIC-Forth and qartus are the same (for both, the test cannot go beyond the 4,0 couple).[98]

[98] To use the program effectively, use option `--stack-size=32767` and `--return-stack-size=16384` (that is the maximum extension for stacks) when invoking qartus on the ack.ft file.

HEX strip, by Baroudi
Vol. III, N. 3 September/October 1981
Another humorous strip: epitome of my nerd life...

(Note: reproduced without permission. If its authors should ask for removal, this strip will disappear in the next version.)

Forth in Laser Fusion, by Lawrence P. Forsley
Vol. III, N. 4 November/December 1981
This volume shows a number of Forth implementations for some industrial and medical fields. The first article, for instance, is a very interesting perspective about an industrial Forth implementation; written with precision, it exposes some interesting results using Forth as the language for the implementation of a laser beam technology. Think this volume dates back to 1978, and think how many years passed from then, and think how early those men did conceived all this...

Implementing Forth based microcomputers at the University of Rochester Medical Center, by Peter H. Helmers
Vol. III, N. 4 November/December 1981
Another Forth specificity. Microcomputers (and also dedicated units) are common today, but in 1981 rather a novelty. The application fields of this medical research were: Ultrasound Diffraction Apparatus (UDA), Vein Mechanics, Pulmonary Microcomputers, X-Ray Scanning System; in all these fields a Forth engine was built, to process input and output. Apparently with success. It's with surprise that the author ends with the Hamletic doubt: "*Anachronism or Portent?*". But further reading shows that "*admittedly, Forth is somewhat limited without such things as a file system or procedural name scoping of variables. Perhaps there should also be less explicit knowledge of addresses, and more system security. Perhaps. But if so, then these things will be evolved as Forth matures.*". Does Forth meet these standards today? I'm not the one that can answer to this.

Data Structures in a telecommunications from end, by John A. Lefor
Vol. III, N. 4 November/December 1981
An excursion within the URTH system, the Forth implementation at the Rochester University Computing Center, since 1977. The study exposes the hardware and software decisions that were made in order to build a telecommunication front end.

Complex Analysis in Forth, by Alfred Clark, Jr
Vol. III, N. 4 November/December 1981
This very interesting article (unfortunately without code) introduces to complex analysis in a very natural way (*natural* from the Forth point of view): `ZDROP`, `ZDUP`, `ZOVER`, `ZROT`, `ZSWAP`, `REZ`, `IMZ`, various math operations like `ZEXP`, `ZSIN`, `ZCOS`, `ZTAN`, even a function `MAP` to map curves and functions, and so forth. Built for the Apple II Forth environment.

HEX strip, by Baroudi
Vol. III, N. 4 November/December 1981
Another humorous strip: holidays are always too short, too distant, too expensive...

(Note: reproduced without permission. If its authors should ask for removal, this strip will disappear in the next version.)

(from letter to the Editor) Julian Date converter, by Peter B. Dunckel
Vol. III, N. 5 January/February 1982 **[julian.ft]**
A converter from a date expressed in `MONTH DAY YEAR` (as single numbers) on the stack (with `YEAR` on top), and returning the Julian Day double number. As stated in the article, the algorithm comes from the U.S. Naval Observatory, via an article appeared in the *Astrophysical Journal Supplement Series*, Vol. 41 No. 3, Nov. 1979, pp 391-2. The comment by the Editor expressed my exact feelings: "*Really slick! But the algorithm would be hard to explain to most people.*".

(from letter to the Editor) Convert to Decimal, by Gregg Williams
Vol. III, N. 5 January/February 1982
A simple and useful converter, called `CVD`, to use decimal numbers when you are in a `BASE` different from 10. For instance, you need to list block `130` and you are in `HEX`; thus `130 LIST` would list block 304! But if you, as suggested by Gregg, define `LIST` as `CVD LIST`, the converter will turn 130 in hexadecimal to 130 in decimal and load the correct block. It works in any `BASE` that accepts the input. You can find `CVD` in the library and you can test it using `REQUIRE` or `ENSURE`.

A technical tutorial: Table lookup examples, by Henry Laxen
Vol. III, N. 5 January/February 1982
An interesting tutorial about methods of creating tables and looking into them. Simple, elegant, useful.

The Game of Reverse, by M. Burton
Vol. III, N. 5 January/February 1982 **[20228-20233]**
The Game of Reverse, written for fig-Forth version 1.15, translated to qartus. Amusement for the brain. The algorithm is simple, yet powerful. And it works perfectly with the same results.

The 31 Game, by Tony Lewis
Vol. III, N. 5 January/February 1982 **[20151-20164]**
The "31 Game". I had to move screen numbers from 51-64 of the original article to 20151-20164, because other sources were in some of the original numbers. Moreover, the program was written using the *micromotion* (c) Forth-79 Version 1.2 for the Apple II, with non-standard screens dimensions (apparently more than 16 lines longer less than 64 characters), so that the transcription was altered both in the source [99] (adding non-standard commands) and in the formatting (to respect the 16*64 format of qartus).

[99] The block 20001 (loaded automatically by the first block 20151) contains special additions for Apple II Forth compatibility.

HEX strip, by Baroudi
Vol. III, N. 5 January/February 1982
Another humorous strip: Forth's way to redefine concepts. And life things.

(Note: reproduced without permission. If its authors should ask for removal, this strip will disappear in the next version.)

A video version of Master Mind, by David Butler
Vol. III, N. 5 January/February 1982 **[20118-20132]**
A wonderful Master Mind game implementation. I had to move screen numbers from 18-32 of the original article to 20118-20132, because other sources were in some of the original numbers. Written in fig-Forth, the conversion was fairly easy. It's an astounding work, because, even if it uses characters, it's nonetheless a graphic program! And I like characters graphic a lot!

Forth Standard Corner, by Robert L. Smith
Vol. III, N. 6 March/April 1982
This article about the Forth-79 Standard `DO..LOOP/+LOOP` works, analyses some problems in the definition of these procedures in the Standard. This article was a great lesson to me. I followed Mr. Smith indication to better qartus `DO`, `LOOP` and `+LOOP`. Read the `DO`, `LOOP` and `+LOOP` entries in this manual.

A techniques tutorial: execution vectors, by Henry Laxen
Vol. III, N. 6 March/April 1982 **[execvec.ft, die.ft]**
This article explains in detail how vectored execution word, by presenting a new enhanced version of `EMIT` (see the didactic file). The example is quite fully compatible with qartus (apart some conversions). Mr. Laxen will publish more articles about this, and in next volumes he will appear again. He was (and maybe still is?) one of the more influent expert in Forth, using the most advanced techniques, presented in a very clear language (clear even for me). Note: the `ASSIGN` case is not compatible with qartus and was not converted.

Charles Moore's BASIC Compiler Revisited, by Michael Perry
Vol. III, N. 6 March/April 1982
This interesting article shows how to emulate a simple BASIC interpreter in Forth. The compiler used for this program is an advanced fig-Forth version with many unknown non-Standard words, and it uses branching based on the Return Stack technique, and as such not compatible with qartus, but it's a wonderful lecture about Forth programming.

A Roundtable on Recursion, by various authors
Vol. III, N. 6 March/April 1982 **[s-ack.ft, recurs.ft]**
A one-page exposition of two recursive techniques. After the foreword by Leo Brody, the Editor, who explains the recursion concept and the working of the `SMUDGE` fig-Forth word, two studies follow: the first by Christoph P. Kukulies (in file `s-ack.ft`), where the Ackermann recursive function is exposed using the `SMUDGE` word, and the second by Arthur J. Smith (in file `recurs.ft`), with his own definition of the `RECURS` word (predating the `RECURSE` Standard word). The `MYSELF` word of the Editor's section differ qualitatively from the qartus version contained in `qlibrary`; finally, the two Brodie words `:R` and `R;`

here defined are not implementable in qartus.

Sieve of Eratostenes [SIC] in Forth, by Mitchell E. Timin
Vol. III, N. 6 March/April 1982 **[20235-20236]**
This program computes the sieve of Eratosthenes for the first 1899 prime numbers in first 8190 integers. I had to change the block numbers, because of other blocks there. Here are Mr. Timin's words: "*The enclosed version of Eratosthenes Sieve was written for an implementation of Timin Forth release 3. I was pleased that it executed in 75.9 seconds, as compared to the 85 seconds of fig-Forth. Mine was run on a 4 MHz 2-80 machine, as were the others in the BYTE magazine article. The speed improvement is primarily due to the array handling capability of Timin Forth release 3.* `FLAGS` *is created with the defining word* `STRING`*;* `n FLAGS` *leaves the address of the nth element of* `FLAGS`*. This calculation occurs in machine code.*". The `STRING` word is not standard, of course. I used `CREATE` and changed the source to adapt to it. Incidentally, on my laptop, the `PRIME-TEST` is run in 385 milliseconds (0.385 seconds), measured through `NTIMER` (contained in the qlibrary.ft file). Amazing.

Turning the stack into local variables, by Marc Perkel
Vol. III, N. 6 March/April 1982 **[argvar.ft]**
A perfect utility that shows the power of Forth. I offer you Mr. Perkel's words: "*Occasionally in writing a definition, I find that I need to do unwieldy stack juggling. For example, suppose you come into a word with the length, width, and height of a box and want to return the volume, surface area, and length of edges. Try it! For this kind of situation I developed my* `ARGUMENTS-RESULTS` *words...*".
I only add that I had to change a bit the code you find in the included file, because the original code didn't work under qartus. So, I judged unfair to put that code in a block which couldn't host the original thought. In any case, Mr. Perkel, you had a very interesting idea!

CASES continued, by various authors
Vol. III, N. 6 March/April 1982
Some pages back, I wrote that in the `CASE` contest, the solution proposed by Mr. Eaker was the nearest match to the current word, or better it **is** the current word. This is plainly shown by a couple of versions (one for the 8080/Z80 computer, by, John J. Cassady, the second for fig-Forth by Alfred J. Monroe, an augmented version) that show how well the Eaker solution was received by the Forth community. The presented cases in this article continue with other proposed versions to solve the `CASE` affair.

Math (monography), by various authors
Vol. IV, N. 1 May/June 1982
My favourite issue! A monography dedicated to math in Fixed-Point. A delightful lecture!

Proposals for Relationals and Multiplication Operators, by Robert L. Smith
Vol. IV, N. 1 May/June 1982
The stainless Mr. Smith offers a tasty appetizer, some fast code and some considerations about operators and signed/unsigned multiplication in Forth.[100] But the generalized concepts remain valid, and this is a good start for a monography![101]

Defining words (part. 1/3), by Henry Laxen
Vol. IV, N. 1 May/June 1982 **[verbs.ft]**
This is the first of a three-parts article about advanced techniques in using `CREATE` and `DOES>`. These articles were a mine of information, and leader texts for the developing of qartus. The word `EMITTER`, being a useful tool, was placed in the `qlibrary` file. Thanks, Mr. Laxen, for these articles.

Fixed-Point Trig by Table-lookup, by John S. James
Vol. IV, N. 1 May/June 1982 **[20032-20035]**
This is a simple but powerful way of dealing with trig in fixed-point math on Forth-79. It gives the

[100] The < code in qartus is built in a strong math-view, so that `-20000 20000 <` returns true; remember also that `M*` and `S->D` are found in `qlibrary.ft`.

functions SIN and COS that calculate their value basing on tables (screen 32), in the range 0°10000 (fixed point numbers with four digits). The application in screen 20035 prints a sine wave in the range 0÷360° (degrees). Load the screens and type WAVE.

Fixed-Point Trig by Derivation, by J. Bumgarner
Vol. IV, N. 1 May/June 1982 **[20036-20037]**
Another technique that uses the Taylor-Maclaurin series to derive an algorithm for the sine, cosine and tangent calculation. The functions SIN and COS return a number in the range 0÷10000; the result is thus a fixed point number (the result multiplied by 10000) with four digits after the dot. TAN instead, is a fixed point number (the result multiplied by 100) in the range 0÷30000; the output is thus not compatible with SIN and COS.

Fixed Point Square Roots, by Klaxon Suralis
Vol. IV, N. 1 May/June 1982 **[20222-20224]**
Calculating square roots of integer numbers (up to 4095) and getting the printing of the square root with three digits of precision? Here is the solution that Mr. Suralis proposes. Being a Forth-79 program, no change was done. I let the redefinition of three words (2DUP, 2DROP and DU<) that Mr. Suralis adds to the program but that qartus has as built-in, but this is harmless. Ah, and I added the word −−> to screens 20222 and 20223, to let the compiler LOAD them all in one turn.

Fractional Arithmetic, by Leo Brodie
Vol. IV, N. 1 May/June 1982 **[20097]**
Here's an interesting point of view: fractions using not decimal values, but scaling the integer with 16384. This has a counterpart: numbers are confined to -2..+2 in range, but you can accomplish calculations with fractions! Here is the example in the article (example that works painlessly in qartus):

```
7 34 /. 23 99 /.  +  .F   <ret> 0.4381
```

(the real value with modern calculators is 0.4382, but you get the idea!) The algorithm is due (in Brodie's words) to Greg Bailey, who conceived the basic design.

The Cordic Algorithm for Fixed-Point Polar Geometry, by Alan T. Furman
Vol. IV, N. 1 May/June 1982 **[20028-20030]**
This article, based on the author's presentation at the March 1982 meeting of FIGGRAPH, shows an interesting approach to polar/rectangular conversions, based on the Cordic Algorithm, in a very few steps (2 blocks). This program runs smoothly on qartus because it's pure Forth-79. Screen 20030 is added, containing the words of the article that show the power of the algorithm. The algorithm is based on the circular behaviour of the integer value, in the range -32768..32767, which is mapped to the circle angles range -180..180, to give a complete 360° elements (this mapping is called 16bitANGLE). This is one of the most interesting articles in *Forth Dimensions*, because it unveils what is the trigonometric calculation done with the Cordic algorithm, that is the basis of all the functions in modern programming languages and in all electronic calculators.

Quadruple Word Simple Arithmetic, by David A. Beers
Vol. IV, N. 1 May/June 1982 **[20400-20412]**
The author with my favourite surname! An interesting article about the development of words for treating quadruple numbers (in the range -2^63..2^63), that is 8 bytes, or 64bit numbers. The article is clear. At page 22, after an ads, a one-page summary (very useful) is given.

Let's go to the saloon!, by McLellan & Brodie
Vol. IV, N. 1 May/June 1982
A humorous cartoon by two Forth programmers, which explains in Forth jargon how this saloon works. In Italy, the sign over the door should recite `18 < ABORT`, but U.S.A. are (were?) more rigorous!

(Note: reproduced without permission. If its authors should ask for removal, this cartoon will disappear in the next version.)

Towards a New Standard, by Robert L. Smith
Vol. IV, N. 2 July/August 1982
In this article, Robert L. Smith introduces the affirmation that Forth-79 vocabularies system is weak, and this problem must be faced. His conclusion: "*There are other designs for vocabulary mechanisms. Almost any of them would be an improvement over FORTH-79. In my opinion it is important that the next Standard have a significant improvement in vocabulary structures.*" Forth-83 will be the answer to this need. Unfortunately for Forth-79...

Defining words II (part. 2/3), by Henry Laxen
Vol. IV, N. 2 July/August 1982 **[class.ft]**
This is the second of a three-parts article about advanced techniques in using `CREATE` and `DOES>`. In this article the `CLASS` word and its correlated words are presented. The program `class.ft` is a transcript of all the listings, suitable for qartus; my advice is to run the program *after* reading the source, and to check the source line by line after run. You'll see the power of this application!

Source Screen Documentation, by Kim Harris
Vol. IV, N. 2 July/August 1982 **[outline.ft]**
In this short article, Kim Harris (of the Laxen & Harris, Inc.) presents a small utility that may reveal useful in treating blocks. `OUTLINE` and `1OUTLINE` may be used to inspect blocks contents if a syntax is respected (the only rule is to start the definitions and comments from the start of the line and the body of definitions from second character on. Simple, isn't it?).

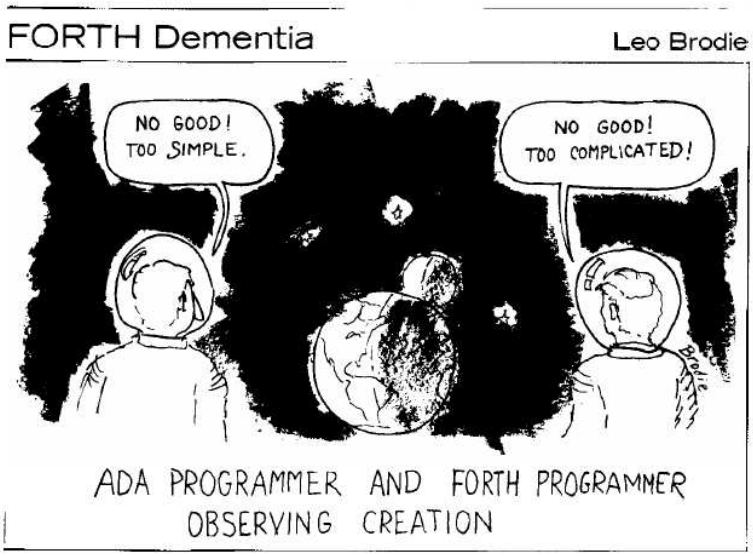
Q T F Quick Text Formatter - Part I, by Leo Brodie
Vol. IV, N. 3 September/October 1982 **[textf.ft, 20073-20075]**
This is the first of two articles by the great Leo, who wrote this program (that you can find it in `textf.ft` with all the corrections for qartus) to print formatted text. The format seems those of the old word assemblers (like Word-Star and the like), and reminds me a bit like groff or Latex. It is a very simple 'language', and it's limited to spacing and indentation, but nonetheless it is another brick to the Forth building.
Note: to execute the program, call `textf.ft` from console, and then execute :

```
20073 load
```

to see the formatted text appear on the monitor. Block 20073 contains some aid-definitions by Leo, while block 20074 is a text block created by me.

Forth-79 Compatible LEAVE for FORTH-83 DO..LOOPS, by Klaxon Suralis
Vol. IV, N. 3 September/October 1982
In this article, Mr. Suralis explains his doubts about the forth-coming Forth-83 implementation of the `LEAVE` in the `DO..LOOP` cycles. His conclusion is "*The existing* **LEAVE** *usage can work with the new* **DO..LOOP**. *We don't have to adopt a direct-jumping* **LEAVE** *in order to reap the benefits of boundary-crossing loop termination. If the immediate* **LEAVE** *is to become standard, let it do so strictly on the basis of its own merits. I sez* [sic] *it LEAVES much to be desired*"

Forth Dementia!, by Leo Brodie
Vol. IV, N. 3 September/October 1982
A humorous cartoon by the great Leo.



(Note: reproduced without permission. If Leo Brodie or the heirs should ask for removal, this cartoon will disappear in the next version.)

Defining words III[102] (part. 3/3), by Henry Laxen
Vol. IV, N. 3 September/October 1982 **[poets.ft]**
This is the third and last of a three-parts article about advanced techniques in using `CREATE` and `DOES>`. In this article the class concept is extended, and it is used to create object classes and sub-classes, that is classes derived by other classes (in this case Mr. Laxen proposes English poets as classes, and their master-pieces are used as a base of data). Unfortunately, this technique does not plainly work with qartus, because of the special use of `AGAIN` which is not standard and thus not available in qartus; the version proposed here is compatible with any Forth-79 Standard, but the process is not replicable (as I suppose the one Mr. Laxed did); look carefully at the original code (reported in comment) and the code developped for qartus: if you can conceive a better and portable solution, you're welcome...

Q T F Quick Text Formatter - Part II, by Leo Brodie
Vol. IV, N. 4 November/December 1982 **[qtf.ft, 21017-21037]**
This is Leo Brodie's editor. The program is long, and I have introduced some changes, in order to make it work under qartus. The first is that CTRL-C cannot be used as a 'character', because it's a system code. So

[101] Curiously not listed in the first page summary...

I turned it to 'c as a command (letter 'c'). The second is that I wrote the codes for the terminal in block 21017 (Leo didn't provide any), and they should work for any modern terminal. The third is that I changed the code for the DELETE key with the code for BACKSPACE (changing from 8 to 127): this seems more adherent to modern requirements. The version on blocks 21018-21037 is complete (you can customize some parts, for instance enable FLUSH by default or enable the snapshot debugger), while the version on file qtf.ft is the Continuous Mode with some transformations (for instance, block 21017, which contains the terminal features, was brought to the end). To start the editor from the base block, type:

```
21018 load
```

from qartus's command line. If you start from block 21017, you'll get an error.

I report here the updated version of the Glossary, that appears at page 25, corrected with the changes I introduced and the unsaid in Brodie's work.

```
                  Quick Text Formatter
                     Editor Glossary

Command Mode
  In this mode, the following keys perform these functions:

i         moves cursor up
j         moves cursor left
k         moves cursor right
m         moves cursor down
J         moves cursor left four characters
K         moves cursor right four characters
a         sets cursor to first character in block
z         sets cursor to last non-blank character in block
c         cuts-off everything to the right of the cursor
n         displays next block
b         displays one block back
e         enters "Enter Mode", allowing you to enter or insert text
r         enters "Replace Mode", allowing you to overwrite text
x         deletes one character, and enters "Delete Mode"
X         the first hit deletes one character and enters "Delete Mode";
          from the second hit on deletes four characters at a time
t         takes the text that was most recently deleted with "x", "X"
          or "c", and inserts it at the current cursor position
RETURN    if in Command Mode, returns to FORTH; otherwise, in any other
          mode, returns to Command Mode
BACKSPACE in Enter Mode, deletes the characters at the left; in
          Delete Mode, cancels the deletion at the left; in any other
          mode, simply moves the cursor to the left. It does NOT store
          characters available to 't'.

Keys 'x' and 'X' works cumulatively, that is they store the whole
body of removed characters, making them available to 't'.
In any mode, a bell sounds when you type an illegal key.
```

Warning: Leo's point of view is not the same of common current editors; he wants his rude editor to be a wide 78 characters space, with a variable number of lines, where you type continuously, and the editor takes care of inserting breaks. A counter appears in the low-right corner of the screen to show how many characters were introduced. It's the user's responsibility to avoid exceeding this limit. I warn you, it's stuff

for tough programmers!

Representation for Logical True, by Robert L. Smith
Vol. IV, N. 4 November/December 1982
In this small article. Mr. Smith exposes the proposal to turn the logical true value as a bit representation (all bits sets, that is -1) rather than as a truth value (one bit set, that is 1), as in Forth-79. The proposal will be accepted, and the next Forth-83 will have -1 as the true value.

Choosing names, by Henry Laxen
Vol. IV, N. 4 November/December 1982
In this article. Mr. Laxen shows his rules for choosing names; I find that his theoris are appreciable still today. The article is full of advices and is an enjoyable lecture, due to his personal wit.

Parameterized CREATE DOES>, by Henry Laxen
Vol. IV, N. 5 January/Febrary 1983 **[param.ft, inform.ft]**
Another interesting article by Henry Laxen, the guru of `CREATE`; in this series, the `CREATE/DOES>` mechanism is pushed towards new boundaries. This article in particular treats the execution of the `DOES>` part choosing the procedure according to a stack structure that holds the execution tokens of the desired procedures, which are effectively pieces of compiled code. The first example (see the file `param.ft`) creates an execution table of the roman numbers printing and, using the index on top of stack, print the corresponding roman: typing for instance `6 ROMAN`, will print `VI`. The second example builds two tables, to print the arabic/roman and arabic/english conversion basing on the two words `ENGLISH` and `ROMAN`.

The code was updated to be used plainly in qartus (for example, `2*` was converted to `2 *` because it is missing in the default database, though present in the library) and, since Mr. Laxen compiler was a "Starting Forth" version Forth (a Forth with some enhancements, as he gently admits in the article), I had to change it a bit, but the version in the included files `param.ft` and `inform.ft` should run smoothly on any Forth-79 compliant compiler.

Personal note: these months between Vol. IV, N. 4 and Vol. V, N. 1 (i.e. between November 1982 and June 1983) include the moment in which I became aware of Forth. Just to let you all know.

FORTH as a Teaching Language, by Albert S. Woodhull
Vol. IV, N. 6 March/April 1983
This article describe an interesting experience by professor Woodhull, Ph.D at the Hampshire College in Amherst, Massachusetts (USA). In three weeks of a January Term, he introduced his students to Forth, which was a complete unknown for all of them. Professor Woodhull explains: "*FORTH has some* [...] *educational advantages* [...]. *Because it has a direct interpretation capability program modules can be debugged much more easily than in a âcompile-onlyâ language like Pascal. Yet FORTH is also a compiler and in a course for advanced students such as I have described FORTH can be used to gain an insight into the compilation process itself...*".

Compilation addresses and Parameter Fields, by Robert L. Smith
Vol. IV, N. 6 March/April 1983
This article throws light into the obscure path that led to the Forth-83 Standard; it actually describes one meeting in which the words `FIND` and `'` (tick) were dissected, and the committee in the end formulated the proposal to adopt `FIND` and `'` for the sole Compilation Addresses in run-time, to introduce `['] ` as the compilation side of `'` and to introduce `>BODY` (at this stage with the temporary name of `BODY`). Illuminating.

Algebraic Expression Evaluation in FORTH, by Michael Stolowitz
Vol. IV, N. 6 March/April 1983

This article introduces one interesting program to evaluate algebraic expressions involving the basic operators * (times), / (divided), + (sum), − (subtraction), < and > (lesser than and greater than), = (equality operator), plus AND, OR and NOT as logical operators. Unfortunately, the code is already in Forth-83 (or in a close enough version), and I couldn't find a translation suitable for qartus. So this remains an exercise for the ones who want to apply.

U/ Bug Fumigated [Technotes, pag. 35], by K.G. Lander
Vol. V, N. 1 May/June 1983
In these few lines (BTW, this is the last issue with Leo Brodie as editor), Mr. Lander critics one Jack Haller for his CODE solution (called U/MOD) for a bug he claimed to have found in U/ (the fig-Forth word for U/MOD, the unsigned division between an unsigned double and an unsigned single number, leaving the unsigned dividend and the unsigned remainder). Mr Lander shows that Haller's solution fails in dealing with the following case:

```
1048676. 65535 U/MOD
```

which returned 31 and 31 rather than 16 and 16. I'm proud to say that qartus behaves exactly as Mr. Lander said U/MOD should do!

Interview with Charles Moore, by Martin Ouverson (the new editor)
Vol. V, N. 2 July/August 1983
A long, pleasant and cheerful interview with the founder of Forth. Themes like productivity, Moore's experiences, idolatry, education in Forth, how he spent his free time, prophetic thoughts about the influence of computers in our lives... His conclusions: "*You have to get inside the world of software, the world of imagination; and you create your own world. You create your own problems and so it is almost frightening the amount of power you've got. I've always had the sort of dream - I've always favored software over hardware. The hardware is the nuisance you have to put up with in order to have software. If you want to talk about a religious aspect of FORTH, it is to say, "*Do you need hardware? Can you conceive of a way of representing these ideas, of making these castles in the air without any particular underpinning?*". I'm sure that [micro]chips will get down to molecular size, but there is still going to be that matter at the core of things. Can you take energy fields, somehow, and weave them to make software? Maybe that's what the layers and layers of operating systems and languages are doing, taking you so far away from the hardware that you forget it is even there*". I completely agree still nowadays.

Recursive Sort on the Stack, by Richard H. Turpin
Vol. V, N. 2 July/August 1983 **[20045]**
This article shows how to sort any numbers on the stack and print them in increasing order (the stack is empty afterwards). The program works perfectly in qartus, in grace of a REQUIRE MYSELF. This program should work painlessly both in any fig-Forth or Forth-79 interpreter (and maybe in all further versions). The version contained in the block 20045 is the one using FLAG.

In `files/` (located in the installation directory) you will find other Forth programs, by various authors (me included), that I added to this package and that don't come from *Forth Dimensions* transcripts. They are:

* `create.ft`: this program shows how `CREATE`d words may or may not inherit the `DOES>` section according to the used creator-word (written by me).
* `defer.ft`: how to implement `DEFER` and `IS` (ANSI 1994) in qartus (written by me).
* `electron.ft`: the SED editor for Forth-79, edited and converted for qartus by Ian Jones. Documentation is in the file `electron.txt`.
* `fix.ft`: how to implement fixed point math (written by Alan Winfield).
* `iftrue.ft`: the library for the three `EXEC-ONLY` words `IFTRUE`, `OTHERWISE` and `IFEND`, described in the Reference Words Section of the Forth-79 Standard (written by me). These words work only if they are all in the same input line.
* `open.ft`: how to use `OPEN-FILE`, `CLOSE-FILE`, `READ-LINE` and `WRITE-LINE` (file dictionary words addenda)
* `stepdown.ft`: simulation of a downward `DO..LOOP`, using the readdressing of the program counter (originally written by Alan Winfield for fig-Forth, adapted for qartus by me).
* `sudokiller.ft`: solution of sudoki grids (written by David Stevenson, and based on Daniele Mazzocchio's sudokiller.asm, adapted for qartus by me).

These programs should be studied before and after execution; you are free to change them and try other solutions suitable for qartus. If they are interesting, send them to me: I'll add them to the next release, if notable.

If you have written programs that qartus can manage, and you want them to be part of the next distribution, send them to me. I'll put them in the package the ones that I will find interesting.

9. Conclusions

The great effort I reversed into qartus, spread among many years, should not be misunderstood. I didn't want to build neither a SwiftForth competitor, nor a gforth 79-sided clone, nor a Polyforth successor, nor an R-Forth cousin or the like. My intents are historic. Forth-79 was a milestone in computer programming and I wanted to honor this. qartus is not to be considered a professional environment, though it can be used profitably by anyone, for learning purposes, for playing, for anything. I hope you like it.

This document and the man page were started in 2011 on a Mac OS X 10.4 Tiger emac and completed in 2020 on a Linux PC with openSuse Leap 15.1.

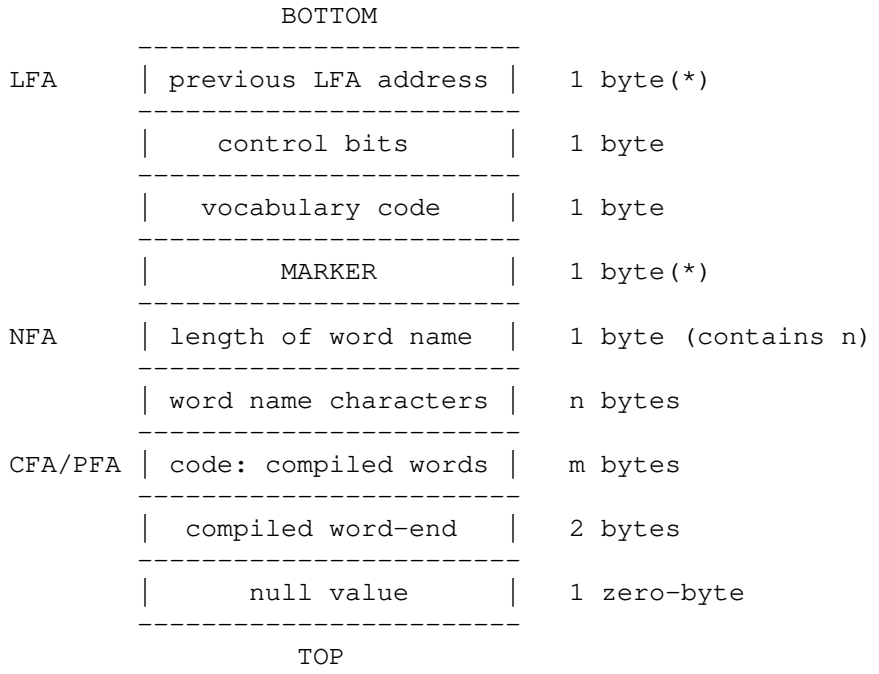
Two things are still to be said.

The first is: I hope you like qartus.

The second is: it's GPL, enjoy! ♥

Appendix: internals

This appendix, largely usable only for enthusiasts, covers some technical data describing the compilation processes of qartus. Each dictionary entry is compiled following the scheme of Fig. 2 found at the beginning of this manual, and here replicated:



(*) in full 32 bits.

Figure #2

This scheme is personal, and does not reflect the classic Forth-79 scheme. My personal design has a strong division between word data (the three data before the MARKER), the word name (the count and the characters following the MARKER, in a classic packed string) and the execution section. This design permits a simple mechanism of execution of a word, which can be compiled by its link address from which all the rest derives. Here follow some explanations about all the sections of the compilation scheme:

* The 'previous LFA address' is a link to the word defined immediately earlier in the dictionary. It's a 32-bit address, used only internally.
* The 'control bits' over a byte define the standard behavior of a word; the byte is structured as follows:

```
no bits set      0   no property
precedence bit   1   word is immediate
smudge bit       2   word is under compilation
buffered bit     4   word is buffered[103]
compile bit      8   word is compile-only
defining bit    16   word is a defining word
addcomp bit     32   word needs special compilation[104]
execution bit   64   word cannot be compiled
```

* The 'vocabulary code' contains a value in the range {0..30} that identifies the vocabulary to which the word belongs to. The vocabulary characteristics are contained in the vocabulary definition word

[102] A buffered word expects characters in the Input Buffer. At present, this characteristic is signaled but not used.

[103] User word have this bit unset. Built-in words have it generally set.

and the vocabulary chaining with the FORTH vocabulary is stored in an inner array, not available by the user.

* The compilation unit follows a two bytes scheme: in the first byte there's a control code, in the second the value/address (in 32 bits) corresponding to the item.

Control codes are listed in fig. #3.

Hex	Decimal	Character[105]	Identifies a	Followed by
49	73	I	16-bit number	2 x 8-bit
44	68	D	32-bit number	4 x 8-bit
64	100	d	16-bit sys. var.	2 x 8-bit
76	118	v	16-bit variable	2 x 8-bit
56	86	V	32-bit variable	4 x 8-bit
6B	107	k	16-bit sys const.	2 x 8-bit
63	99	c	16-bit constant	2 x 8-bit
43	67	C	32-bit constant	4 x 8-bit
55	220	Ü	user word	1 x 32-bit
5F	230	æ	built-in word	1 x 16-bit

Figure #3

* Some words are compiled along with the data they treat. For instance, the ." word is stored as _ followed by the ." numeric code, followed by the packed string which is the object of compilation.
* The word-end for built-in words is marked by an empty byte.
* The word-end for user words is marked by the compiled built-in code (which is 0xFF) plus an empty byte.

As an example, take a look at figs. #4, #5 and #6, where the built-in word DUP, the built-in variable SCR and a user variable YAP are compiled. In the first, since there is no parameter, the PFA has no meaning; in the second and third (supposing that SCR holds the value of 100 and YAP 2100), since there is no code, the CFA has no meaning[106].

	MEM.	CODE	DESCRIPTION
LFA	092E	00000922	link to previous word (32-bit)
	092F	00000020	control bits (special compilation)
	0930	00000000	vocabulary code (0 = FORTH)
	0931	0AACAADA	marker (32-bit)
NFA	0932	00000003	number of characters in name
	0933	00000064	letter d
	0934	00000075	letter u
	0935	00000070	letter p
CFA	0936	0000005F	control code for built-in word
	0937	0000003B	DUP's code
	0938	00000000	null value

Figure #4: DUP

[104] I introduced these characters values identifiers to make them immediately detectable in a memory DUMP.

[105] In the next figures and in all the other, addresses may vary in your configuration. Take them only as examples.

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 004D   00000041   link to previous word (32-bit)
    004E   00000000   control bits
    004F   00000000   vocabulary code (0 = FORTH)
    0050   0AACAADA   marker (32-bit)
NFA 0051   00000003   number of characters in name
    0052   00000073   letter s
    0053   00000063   letter c
    0054   00000072   letter r
    0055   00000064   control code for system variable
PFA 0056   00000064   16-bit low part
    0057   00000000   16-bit high part
    0058   00000000   null value
```

Figure #5: SCR

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 1508   000014F8   link to previous word (32-bit)
    1509   00000000   control bits
    150A   00000000   vocabulary code (0 = FORTH)
    150B   0AACAADA   marker (32-bit)
NFA 150C   00000003   number of characters in name
    150D   00000079   letter y
    150E   00000061   letter a
    150F   00000070   letter p
    1510   00000076   control code for user variable
PFA 1511   00000034   YAP high part
    1512   00000008   YAP low part
    1513   00000000   null value
```

Figure #6: YAP (user variable)

Now, let's consider the following small program:

```
: MYDUP   DUP DUP ;
: CODE    MYDUP + ;
```

Previous words are compiled this way:

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 14E7   000014B8   link to previous word (32-bit)
    14E8   00000000   control bits
    14E9   00000000   vocabulary code (0 = FORTH)
    14EA   0AACAADA   marker (32-bit)
NFA 14EB   00000005   number of characters in name
    14EC   0000006D   letter m
    14ED   00000079   letter y
    14EE   00000064   letter d
    14EF   00000075   letter u
    14F0   00000070   letter p
CFA 14F1   0000005F   control code for built-in word
    14F2   0000003B   first DUP's code
    14F3   0000005F   control code for built-in word
    14F4   0000003B   second DUP's code
    14F5   0000005F   control code for built-in word
    14F6   000000FF   code for end word
    14F7   00000000   null value
```

Figure #7: MYDUP (user program)

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 14F8   000014E7   link to previous word (32-bit)
    14F9   00000000   control bits
    14FA   00000000   vocabulary code (0 = FORTH)
    14FB   0AACAADA   marker (32-bit)
NFA 14FC   00000004   number of characters in name
    14FD   00000063   letter c
    14FE   0000006F   letter o
    14FF   00000064   letter d
    1500   00000065   letter e
CFA 1501   00000055   control code for user word
    1502   000014E7   address containing MY-DUP address (32-bit)
    1503   0000005F   control code for built-in word
    1504   00000097   + code (plus)
    1505   0000005F   control code for built-in word
    1506   000000FF   code for end word
    1507   00000000   null value
```

Figure #8: CODE (user program)

The next figure is an example of a CREATE word, as follows:

```
CREATE ARRAY

        MEM.   CODE       DESCRIPTION
      -------------------------------------------------
      LFA 1514   00001508   link to previous word (32-bit)
          1515   00000000   control bits
          1516   00000000   vocabulary code (0 = FORTH)
          1517   0AACAADA   marker (32-bit)
      NFA 1518   00000005   number of characters in name
          1519   00000061   letter a
          151A   00000072   letter r
          151B   00000072   letter r
          151C   00000061   letter a
          151D   00000079   letter y
      CFA 151E   0000005F   control code for built-in word
          151F   000000EB   code for no-op (future usage)
          1520   00000049   control code for 16-bit integer
          1521   00000026   PFA address low part
          1522   00000015   PFA address high part
          1523   0000005F   control code for built-in word
          1524   000000FF   code for end word
          1525   00000000   null value
      PFA 1526   00000000   PFA address of CREATEd word
```

Figure #9: ARRAY (user structure)

ALLOT, used after a CREATE, simply extends the space dedicated to the word, starting from the actual start of the CREATEd word (the address 1526 in the example). The address is returned by the CREATEd word in virtue of the 16-bit integer code hardwired in the CREATEd word's code.

A note is worthwhile here: CREATE, as stated from the Standard, does not allocate any parameter field memory. So, a line like:

```
CREATE ARRAY
ARRAY .
```

prints the value of HERE; any subsequent compilation would start from the parameter field returned by ARRAY; that's why compiling words like ALLOT, , (comma) and C, (C-comma) are used, because they advance HERE and 'protect' data.

If CREATE is used in conjunction with DOES>, the compilation scheme is so structured; suppose we have a defining word like this:

```
: HELPER
   CREATE  ." Helper created"
      DOES> ." How can I help?"
;

HELPER ROOM
```

When HELPER ROOM is invoked, the string "Helper created" is printed. When then ROOM is invoked, the

string "How can I help?" is printed. The two words HELPER and ROOM are compiled as follows:

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 0E33   000009F6   link to previous word (32-bit)
    0E34   00000010   control bits
    0E35   00000000   vocabulary code (0 = FORTH)
    0E36   0AACAADA   marker (32-bit)
NFA 0E37   00000006   number of characters in name
    0E38   00000068   letter h
    0E39   00000065   letter e
    0E3A   0000006C   letter l
    0E3B   00000070   letter p
    0E3C   00000065   letter e
    0E3D   00000072   letter r
CFA 0E3E   0000005F   control code for built-in word
    0E3F   00000027   code for CREATE
    0E40   0DAAFAAC   DOES> marker (32-bit)
    0E41   00000E55   address of DOES> part (32-bit)
    0E42   0000005F   control code for built-in word
    0E43   00000036   code for ." (dot-quote)
    0E44   0000000E   string counter (14 characters)
    0E45 ♦ 00000E52    String "Helper created"
    0E53   0000005F   control code for built-in word
    0E54   000000ED   code for end word marked by DOES>
                      START OF DOES PART
    0E55   0000005F   control code for built-in word
    0E56   00000036   code for ." (dot-quote)
    0E57   0000000F   string counter (15 characters)
    0E58 ♦ 00000E66    String "How can I help?"
    0E67   0000005F   control code for built-in word
    0E68   000000FF   code for end word
    0E69   00000000   null value
```

Figure 10: HELPER (defining word)

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 0E6A   00000E33   link to previous word (32-bit)
    0E6B   00000000   control bits
    0E6C   00000000   vocabulary code (0 = FORTH)
    0E6D   0AACAADA   marker (32-bit)
NFA 0E6E   00000004   number of characters in name
    0E6F   00000072   letter r
    0E70   0000006F   letter o
    0E71   0000006F   letter o
    0372   0000006D   letter m
CFA 0E73   0000005F   control code for built-in word
    0E74   000000EB   code for no-op (future usage)
    0E75   00000049   control code for 16-bit integer
    0E76   0000008D   PFA address low part
    0377   0000000E   PFA address high part.
    0E78   0000005F   control code for built-in word
    0E79   00000036   code for ." (dot-quote)
    0E7A   0000000F   string counter (15 characters)
    0E7B ♦ 00000E89   String "How can I help?"
    0E8A   0000005F   control code for built-in word
    0E8B   000000FF   code for end word
    0E8C   00000000   null value
PFA 0E8D   00000000   PFA address of CREATEd word
```

Figure 11: ROOM (word defined through HELPER)

Next, as an example of a 32-bit item, here's the decompilation of a `2CONSTANT` element, obtained via

```
3.14159265 2CONSTANT PI
```

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 1526   00001514   link to previous word (32-bit)
    1527   00000000   control bits
    1528   00000000   vocabulary code (0 = FORTH)
    1529   0AACAADA   marker (32-bit)
NFA 152A   00000002   number of characters in name
    152B   00000070   letter p
    152C   00000069   letter i
    152D   00000043   control code for 32-bit constant
PFA 152E   000000B9   32-bit high part: 16-bit low part
    152F   00000012   32-bit high part: 16-bit high part
    1530   FFFFFFA1   32-bit low part: 16-bit low part
    1531   FFFFFFB1   32-bit low part: 16-bit high part
    1532   00000000   null value
```

Figure 12: PI (user 32-bit constant)

See now the decompilation of a peculiar word, the `RECURSE` word contained in `qlibrary.ft`.[107]

[106] The `RECURSE` word contained in qlibrary.ft is not portable, because it uses the peculiar word `32,` to store in a unit a 32-bit number (operation permitted by qartus given its memory structure). The disassembling here reported should so be taken as a mere example.

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 0E40   00000CBB   link to previous word (32-bit)
    0E41   00000001   control bits (immediate)
    0E42   00000000   vocabulary code (0 = FORTH)
    0E43   0AACAADA   marker (32-bit)
NFA 0E44   00000007   number of characters in name
    0E45   00000072   letter r
    0E46   00000065   letter e
    0E47   00000063   letter c
    0E48   00000075   letter u
    0E49   00000072   letter r
    0E4A   00000073   letter s
    0E4B   00000065   letter e
CFA 0E4C   0000005F   control code for built-in word
    0E4D   0000002D   code for DECIMAL
    0E4E   00000044   control code for 32-bit integer
    0e4f   00000000   integer high part - high part
    0e50   00000000   integer high part - low part
    0e51   00000055   integer low - high part
    0e52   00000000   integer low part - low part
    0E53   0000005F   control code for built-in word
    0E54   00000001   code for 32,
    0E55   00000064   control code for system variable
    0E56   000000C1   address of LAST
    0E57   0000005F   control code for built-in word
    0E58   00000065   code for @
    0E59   00000055   control code for user word
    0E5A   00000E2E   address of S->D (previously loaded)
    0E5B   0000005F   control code for built-in word
    0E5C   00000001   code for 32,
    0E5D   0000005F   control code for built-in word
    0E5E   000000FF   code for end word
    0E5F   00000000   null value
```

Figure #13: RECURSE (user program)

As a final example, here it is the decompilation of an user vocabulary, created through the invocation of:

```
VOCABULARY TEST
```

```
    MEM.   CODE       DESCRIPTION
-------------------------------------------------
LFA 0BCD   00000BC1   link to previous word (32-bit)
    0BCE   00000000   control bits
    0BCF   00000000   vocabulary code (0 = FORTH)
    0BD0   0AACAADA   marker (32-bit)
NFA 0BD1   00000004   number of characters in name
    0BD2   00000074   letter t
    0BD3   00000065   letter e
    0BD4   00000073   letter s
    0BD5   00000074   letter t
CFA 0BD6   0000005F   control code for built-in word
    0BD7   000000E9   code for (VOCABULARY)
    0BD8   0CEAFACC   vocabulary marker (32-bit)
    0BD9   00000002   vocabulary number
    0BDA   0000005F   control code for built-in word
    0BDB   000000FF   code for end word
    0BDC   00000000   null value
```

Figure #14: TEST (user vocabulary)

A vocabulary is not a common word, in qartus; it is a special word which contains the compiled number of the vocabulary and the code of a run-time procedure, called `(VOCABULARY)` that, when executed, will set the compiled number as `CONTEXT`. Thus, it has no PFA part because the section from CFA to the word-end is a special code section that should not be modified at all.

****

As you have certainly detected, the `DOES>` section of a word is copied, not referenced, to the created word. This assures that the created word is a complete and independent word. This feature is thus a design feature. Moreover, as you have certainly noted, in qartus the word's name is stored in full.

I think the previous design features have some advantages and some disadvantages:

- Advantages are an easier search, an easier compilation, and an easier understanding of programs, names positions and addresses when dumping memory; sometimes this turns to be essential during debugging.

- Disadvantages consist in a small delay in execution, while checking for words names, and (probably) a small increment of memory occupation.

Something I can live with.

To end up, here is the default output (from a fresh start) of the `0 INFO` execution word (`<user>` stands for the user's name).

```
qartus running mode:        Forth-79

qartus memory settings (in bytes):
Memory sectors              Width [bytes]     Start [addr]
-----------------------------------------------------------
Start of memory                    1               0 (reserved)
Program memory                 55807               1 (available 51909)
Memory in use                   3898
Terminal input buffer           1024           55808
WORD buffer                     1024           56832
Scratch area                     512           57856
Screen buffer 1                 1024           58368
Screen buffer 2                 1024           59392
Screen buffer 3                 1024           60416
Data stack                      2048           61440
Return stack                    2048           63488
-----------------------------------------------------------
End of memory                                  65535

qartus status:
-------------
last compiled word:   none
last accessed block:  none
last accessed buffer: none
words compiled so far: 313 (no user words yet)

qartus limits and features:
--------------------------
Maximum strings size: 255
Size of PAD: 84 (200 bytes after HERE)
Type of division: symmetric
Maximum dimension of a character: 8 bits
  Default ASCII range is 0..127 (7 bits)
  Range 128-255 is O.S. dependent
Maximum string length: 255 characters
Screen/Block default size: 1024 bytes
Maximum number of word lists usable in the search order: 31
Largest usable unsigned double number: 4294967295
Largest usable unsigned number: 65535
Largest positive usable signed double number: 2147483647
Largest negative usable signed double number: -2147483648
Largest positive usable signed number: 32767
Largest negative usable signed number: -32768

system dictionary status:
------------------------
list of immediate words: 32
--> endcase endof of case +loop loop do until end again while repeat
begin then endif else if ascii ' [ ] ; (sliteral) ." dliteral literal
does> editor forth ) (

list of compile-only words: 50
(endcase) endcase (endof) endof (of) of (case) case (+loop) +loop (loop)
loop (do) do (until) until end (again) again (while) while (repeat) repeat
(begin) begin leave j i (then) then endif (else) else (if) if (vocabulary)
; (sliteral) [compile] (compile) compile exit dliteral literal >r r> r@
(does>) does> (word)

list of defining words: 6
vocabulary 2constant 2variable constant variable create

list of executable-only words: 14
edit load-system save-system include search ensure require debug trace
exec-only compile-only forget : immediate

user dictionary status:
----------------------
list of user words: 0
```

```
list of user vocabularies: 0

system directories and files:
----------------------------
system directory: /home/<user>/.qartus/
blocks directory: /home/<user>/.qartus/blocks/
current linked library: /home/<user>/.qartus/qlibrary.ft
resource file: /home/<user>/.qartus.rc (disabled)
```

Table of Contents

1. **General notes** . . . 3
1.1. Introducing qartus . . . 3
1.2. The documents . . . 4
1.3. Definition of terms . . . 8
1.4. Fulfilled requirements . . . 17
1.5. The memory model . . . 17
1.6. The run-time engine . . . 20
1.6.1. The execution model . . . 20
1.6.2. The Program Counter . . . 20
1.6.3. The 'readline' facility . . . 21
1.6.4. Program interruption . . . 21
1.7. System constants . . . 22
1.8. System variables . . . 22
1.9. Error management . . . 23
1.9.1. Default Error Management . . . 23
1.9.2. User Error management . . . 24
1.10. Using BLOCK and BUFFER . . . 25
1.10.1. Using BLOCK . . . 25
1.10.2. Using BUFFER . . . 25
1.10.3. Choosing specific blocks directories with USE . . . 26
1.11. Extensions for File-treatment . . . 26
1.12. Using computer ports . . . 27
1.13. Making sounds with qartus . . . 28
1.13.1. Using BELL . . . 28
1.13.2. Using (SOUND) . . . 28
1.14. Pipes with qartus . . . 29
1.14.1. Reading a pipe . . . 29
1.14.2. Writing to a pipe . . . 29
1.15. Super-immediate words . . . 29
2. **Some discussions about Forth-79** . . . 30
2.1. A word about FIND and ' (tick) . . . 30
2.2. DO, LOOP and +LOOP problems . . . 30
2.3. Something about EXECUTE . . . 32
2.3.1. The 'what' side: Execution of the compilation address . . . 32
2.3.2. The 'how many' side: Multiple or single execution . . . 32
2.4. LEAVE or not leave? . . . 33
2.5. VOCABULARY problems . . . 33
2.5.1. Chaining . . . 33
2.5.2. Immediacy . . . 34
2.6. A word about LITERAL (and DLITERAL) . . . 34
3. **Words Reference** . . . 35
3.1. Notation in the Reference Manual . . . 35
3.2. The Dictionary . . . 36
3.3. The Line Editor in detail . . . 79
3.3.1. Some Line Editor techniques . . . 83
3.4. The Screen Editor in detail . . . 84
3.4.1. Status lines . . . 85
3.4.2. Screen Editor commands . . . 85
3.5. fig-Compatibility . . . 87
3.6. Service words . . . 88

3.7. The integration file qlibrary.ft . . . 90
4. **The built-in debugger** . . . 91
4.1. Using DEBUG . . . 91
4.2. Using TRACE . . . 92
4.3. Debug Commands . . . 92
5. **Differences with the Forth-79 Standard** . . . 94
5.1. Changes . . . 94
5.1.1. Word sets . . . 94
5.1.2. --> . . . 94
5.1.3. dot-quote ." . . . 94
5.1.4. BASE . . . 94
5.1.5. BLOCK and BUFFER . . . 94
5.1.6. CONTEXT . . . 95
5.1.7. CURRENT . . . 95
5.1.8. DPL . . . 95
5.1.9. FORGET . . . 95
5.1.10. LITERAL and DLITERAL . . . 95
5.1.11. PAGE . . . 95
5.1.12. WORD . . . 95
5.2. Enhancements . . . 96
5.2.1. Memory . . . 96
5.2.2. The (comment . . . 96
5.2.3. CREATE . . . 96
5.2.4. DOES> . . . 96
5.2.5. EMIT . . . 96
5.2.6. TYPE . . . 97
5.2.7. WORD . . . 97
6. **Forth-83 and beyond (how evolution has won)** . . . 98
6.1. The Compilation Address . . . 98
6.2. To be or NOT **to be** . . . 98
6.3. FIND and ' (tick) . . . 98
6.4. Unsigned index arguments for memory commands . . . 98
6.5. Removal of STATE-dependent words . . . 99
6.6. Improved DO..LOOP . . . 99
6.7. Division type . . . 99
6.8. Zero-based PICK and ROLL . . . 99
6.9. Improved WORD . . . 99
6.10. Renaming of some words . . . 99
7. **Conversion of fig-Forth programs** . . . 100
7.1. Can I say a WORD? . . . 100
7.2. +LOOP limits . . . 100
7.3. Using VARIABLE properly . . . 101
7.4. A SIGN of destiny . . . 101
7.5. How to FIND it? . . . 102
7.6. A VOCABULARY lookup . . . 102
7.7. Where is it? . . . 103
8. **My favourite FD Articles** . . . 104
9. **Conclusions** . . . 119
Appendix: internals . . . 120