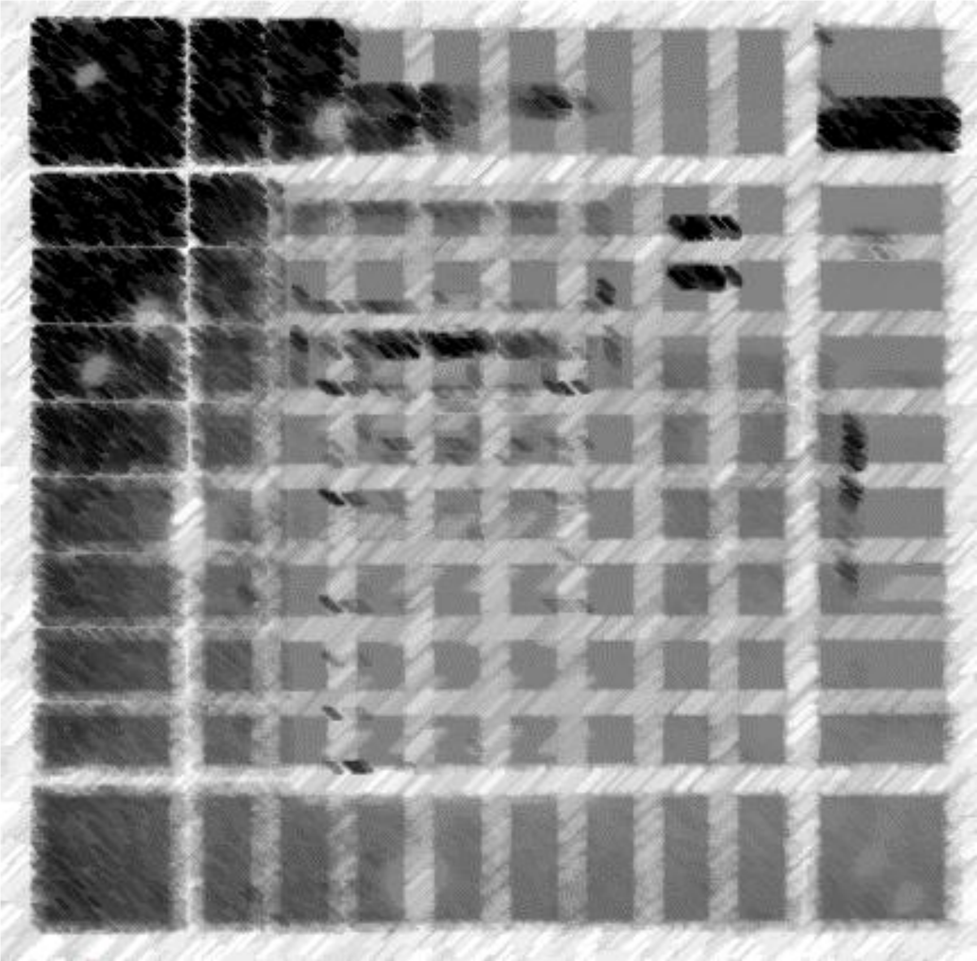
FORTH_41+ Module

FORTH for the HP-41



Forth-41 was programmed by Serge Vaudenay,
with collaboration of Stéphane Barizien and Jean-Jacques Dhénin

Document date: December 15, 2020

This compilation revision 1.1.2

Acknowledgments.-

First and foremost, thanks to Serge Vaudenay for creating the FORTH Module. An excellent example of creative and clever programming all way through.

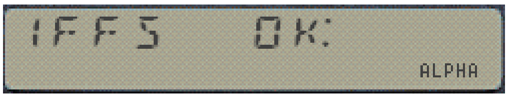
Thanks to Stéphane Barizien and Jean-Jacques Dhénin, who wrote the READEM/WRITEM functions, also available in the PANAME ROM. I believe they may have also contributed to the FORTH code but there's no official record of it.

Thanks to Greg McClure for his valuable suggestions on the MCODE reverse-engineering process.

Emmanuel Compes created the V41 Bitmap shown in page #5 and wrote the original QRG this manual is based on.

Screen captures taken from V41, Windows-based emulator developed by Warren Furlow.
See www.hp41.org

Foreword


1FF5 OK:

ALPHA

This document includes notes and observations made during the investigation of the FORTH41 module. It builds on the rudimentary manual available for it, which was very sketchy and incomplete, only a painfully short approximation to the usage and description of the functionality.

A few initial remarks :

- This is a special 8k module that uses page #4 for the FORTH routine library and other housekeeping chores. The upper page can occupy any of the external addresses on the HP-41 bus (from 8 to F), but should not use pages #6 and #7 for printer and HP-IL compatibility. If the CX or the Time module is used, then page #5 is also reserved and shouldn't be used either.

- The module provides a separate environment for FORTH operation, replacing the standard HP-41 operating system – but allows calls to HP-41 programs and functions – both in the standard mainframe and other application ROMS.

- Setting User **Flag 18** activates the "Terminal Console" mode, capturing most of the command line activity into the HP-IL output device (supposedly a monitor but also works on the printer on V41 via ILPER). Notably the output also includes the compilation commands, very useful for troubleshooting. Note that UF 15/16 have no impact on the FORTH environment!

- Programs and definitions survive power-cycling and switching the machine off – ***but the data in the stack does not***. The FORTH mode will be active when you switch it on, and you can resume work. It's also possible to leave the FORTH mode (BYE) to use the calculator in its native form, and then resume using FORTH, but the stack data will be lost as well.

- FORTH41+ includes a set of 135 words in the ROM dictionary, most of them primitives but some secondaries as well, based on the primitives. It's possible to define more words in RAM using the appropriate commands.

- FORTH41 can only operate on 16-bit integer words, from 0 to FFFF (unsigned) ; or from -7FFF to 7FFF (signed). Some primitives operate on 8-digit bytes as well as 16-bit integers.

- The command line input is limited by the length of the ALPHA registers, i.e. 24 chars max. However up to five "full" command lines can be manually loaded in the INPUT buffer RAM memory.

- Programs can be typed manually or loaded from ASCII files in Extended Memory. The module includes a Text File Editor program (EDA) with support for the special FORTH syntax characters. This is an important detail that rules out the unmodified ASCII File Editor (ED) in the CX – but you can use the enhanced version (ED+) available in the WARP module or on the SY-41CL and DM41X machines.

- The module also includes two functions to write/read the complete contents of the X-Memory configuration in an external HP-IL Mass Storage drive. Interestingly, these are a direct port from the PANAME ROM – the *French connection* at work. These functions take up a large amount in the ROM and could be removed to allow for extensions of the vocabulary if desired.

Despite the design limitations this module is a very complex MCODE work with a very ambitious reach and scope. Unfortunately, the lack of complete information restricts the usage of the module to exploration and research, as opposed to practical applications. There are multiple and very basic areas of functionality still unexplored, as progress is slowed down by the lack of any specifications or manual. The buffer design and structure in particular is almost completely unknown, which has direct impact on the vocabularies and RAM arrangement. A work in progress, time will tell how far we get…

Introduction

Another realization of Serge Vaudenay (T270) after the ROMSV01, the FORTH41 module is 8 kBytes and occupies pages 4 and an external one (8 – F) of the HP-41. It includes 62 functions and its language is rich with 135 primitives. CATALOG 2 thus presents 61 words or functions divided into three unequal parts :

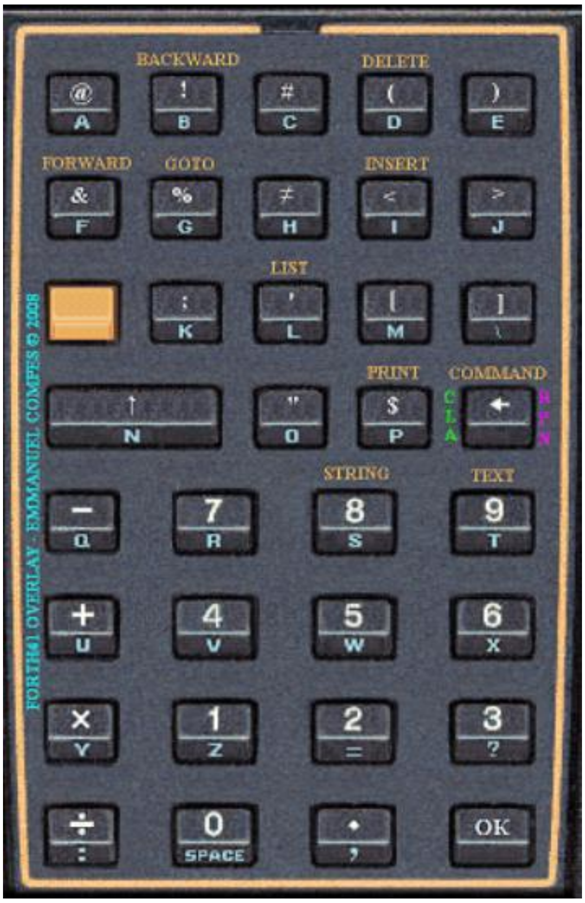
- -FORTH 41+ These are FORTH words specific to the HP-41.
- -COMPILER The following words are all preceded by **}-** . These words are in fact programs used by the compiler. They are called in the definitions of the secondaries.
- -PRIMITIVES FORTH-79 words (except AREDIT) accessible from the RPN environment.

CAT'2 Functions of FORTH41

-FORTH 41+	XROM 09,00	not applicable, name of the ROM
CRFRAM	XROM 09,01	creates a FORTH memory of X registers by destroying the buffers
"EDA	XROM 09,02	edits an alpha file (creates it from X registers if needed)
FEX	XROM 09,03	validates the alpha register in FORTH
FORTH	XROM 09,04	switches to FORTH mode
FRAMCHK	XROM 09,05	replaces the FORTH memory
FTOX	XROM 09,06	stack transfer FORTH to register X
READEM	XROM 09,07	reads FORTH memory from mass memory
WRTEM	XROM 09,08	saves FORTH memory to a mass storage device
XTOF	XROM 09,09	stack Transfer register X to FORTH
-COMPILER	XROM 09,10	not applicable, physical separation in the catalog
}-**CALL**	XROM 09,11	Calls a FOCAL program by its global Label name
}-**COMP**	XROM 09,12	Compiles text between spaces
}-**DATA**	XROM 09,13	
}-EX	XROM 09,14	
}-EXB	XROM 09,15	Executes MCODE on RAM buffer
}-EXO	XROM 09,16	Executes FOCAL code at address in two bytes after call
}-EXP	XROM 09,17	Executes MCODE at address in two bytes after call
}-MLP	XROM 09,18	Executes MCODE just below the call
}-RNT	XROM 09,19	
-PRIMITIVES	XROM 09,20	Selected Primitives on CAT-2
AREDIT	XROM 09,21	equivalent to LINPUT on the HP-71B
@	XROM 09,22	Two-byte recall
!	XROM 09,23	Two-byte store
C@	XROM 09,24	one byte recall
C!	XROM 09,25	one byte store
+!	XROM 09,26	adds at given address
?	XROM 09,27	display a cell content
>R	XROM 09,28	to return stack
R>	XROM 09,29	from return stack
DUP	XROM 09,30	duplication of the top stack level
DROP	XROM 09,31	drops one level of the stack
SWAP	XROM 09,32	swaps the first two stack levels
OVER	XROM 09,33	puts second stack level on top
ROT	XROM 09,34	rotates the first three stack levels
}-	XROM 09,35	Print top level stack
U}-	XROM 09,36	Print Unsigned
EMIT	XROM 09,37	sends ASCII char to display
TYPE	XROM 09,38	sends a string to the display from a given address
#	XROM 09,39	digit formatting string
+	XROM 09,40	addition
-	XROM 09,41	subtraction
*	XROM 09,42	multiplication
/	XROM 09,43	integer division
***/**	XROM 09,44	multiplication followed by division

MOD	XROM 09,45	Modulo function
/MOD	XROM 09,46	Quotient remainder
ABS	XROM 09,47	Absolute value
NEGATE	XROM 09,48	changes the sign
=	XROM 09,49	equal test
<	XROM 09,50	inferior test
>	XROM 09,51	superior test
0=	XROM 09,52	tests if zero
0<	XROM 09,53	tests if less than zero
0>	XROM 09,54	tests if more than zero
1+	XROM 09,55	adds one
1-	XROM 09,56	subtracts one
2+	XROM 09,57	adds two
2-	XROM 09,58	subtracts two
AND	XROM 09,59	logical AND
OR	XROM 09,60	logical OR
NOT	XROM 09,61	logical NOT
BRANCH	XROM 09,62	Relative jump
0BRANC	XROM 09,63	Conditional relative jump

The figure below shows the FORTH keyboard with the location of the special characters needed for FORTH syntax. Note also the shortcuts to the control commands from the File Editor program (**EDA**). You can download the file for V41 here: http://www.hp41.org/LibView.cfm?Command=View&ItemID=637



Details of the 135 primitives of FORTH-41

All commands can be input on the command line when the FORTH mode is active. A number of primitives are also in CAT 2 so they can be used directly (no need to use FEX); provided of course that the FORTH memory is in its place. Some of the primitives are used to read the user variables and are listed below on grey background. Note also the four new additions in this revision on cyan highlight.

1	**-**	subtraction
2	**!**	Stores a number at adr
3	**#**	generate from double
4	**#>**	end of formatting string
5	**#S**	remaining of a number
6	**(**	Starts comment
7	**)**	End comment
8	*****	product n1 * n2
9	***/**	Multiply and division
10	**,**	Places n in dictionary.
11	**.**	displays top of stack
12	**.STATE**	Compile state flag
13	**/**	division n1/n2
14	**/MOD**	(n1 n2 -- rem quot)
15	**:**	starts variable define
16	**;**	ends variable define
17	**?**	combination of @ **.**
18	**?DUP**	does DUP if n#0
19	**@**	Fetches n from adr
20	**[**	Begins interpreting.
21	**]**	Ends interpreting,
22	**‘**	search for word addr.
23	**+**	(n1, n2 -- sum)
24	**+!**	Adds 1 to n at adr
25	**+LOOP**	repeats Do/Loop +n
26	**<**	(n1, n2 – bool)
27	**<#**	display formatting start
28	**=**	(n1, n2 – bool)
29	**>**	(n1, n2 – bool)
30	**>IN**	top of input block
31	**>MM**	Converts to MM format
32	**>R**	moves n to RTN stack
33	**0**	enters zero
34	**0<**	tests if <0
35	**0=**	tests if =0
36	**0>**	tests if >0
37	**0BRANCH**	conditional branch
38	**1-**	subtracts one from top
39	**1+**	adds one to top level
40	**2-**	subtract 2 from top
41	**2+**	add 2 to top level
42	**2***	doubles stack top level
43	**2/**	divides top level by 2
44	**ABORT**	initializes FORTH file
45	**ABS**	absolute value
46	**ALLOT**	Places n bytes in dictr.
47	**ALPHA**	ALPHA Status
48	**AND**	Bitwise AND
49	**AREDIT**	like LINPUT on the 71
50	**BASE**	ADR of number base
51	**BEGIN**	Begins a loop
52	**BRANCH**	(addr, distance)
53	**BUF0**	??? Unknown
54	**BYE**	*return to HP41 mode*
55	**C!**	Stores byte at addr
56	**C,**	like **,** but on a byte
57	**C@**	Fetches byte from adr
58	**CALL**	*call to HP41 program*
59	**CD,**	like C, decrement adr
60	**CFLAG**	Clears User Flag
61	**CMOVE**	Move bytes btw. adr
62	**COMPILE**	Compiles next word
63	**CONSTANT**	Creates a constant
64	**CONTEXT**	user variable
65	**COUNT**	Unpacks a string
66	**CR**	Carriage Return
67	**CRBUF**	Creates Buffer
68	**CREATE**	Creates new command
69	**CRFILE**	(flsize) Data File
70	**CTRL**	*control sequence*
71	**CURRENT**	user variable
72	**D,**	Like , decrement adr
73	**DECIMAL**	sets DEC mode
74	**DEFINITIONS**	???
75	**DEPTH**	Shows stack depth
76	**DESTROY**	*destroys buffers*
77	**DO**	Begins a DO…LOOP
78	**DOES>**	Provides instructions…
79	**DROP**	Drops stack
80	**DUP**	Duplicates top level
81	**EDA**	*Text editor (flname$)*
82	**ELSE**	(bool) false
83	**EMIT**	(char#)
84	**EXIT**	Returns from current w
85	**FIND**	search in dictionary
86	**FLAG**	Gets UFlag status
87	**FORGET**	Removes (var$)
88	**FTOX**	*FORTH to X-reg*
89	**GROW**	??? Unknown
90	**HERE**	next dictionary adr
91	**HEX**	sets HEX mode
92	**HOLD**	ASCII char in string
93	**I**	puts loop index in stx
94	**IF**	(bool) true
95	**IMMEDIATE**	Marks the last def.
96	**ITYPE**	??? Unknown
97	**J**	outer loop index to stk
98	**KEY**	Reads Input character
99	**LEAVE**	exits from Do/Loop
100	**LOADF**	Loads (flname$)
101	**LOOP**	End of DO…LOOP
102	**M1-**	subtracts 1 from adr
103	**M1+**	adds 1 to adr
104	**M2-**	subtracts 2 from adr
105	**M2+**	adds 2 to adr

106	**MM>**	Converts to ADR frmat
107	**MOD**	(n1 n2 -- rem)
108	**NEGATE**	Two’s complement
109	**NOT**	logical NOT
110	**OCTAL**	Sets Octal mode
111	**OR**	Bitwise OR
112	**OVER**	Copies 2nd to top level
113	**PAD**	address of PAD area
114	**PARM**	Defines parameter
115	**QUIT**	sets light-sleep mode
116	**R>**	moves from RTN stack
117	**R0**	internal FORTH flag
118	**REPEAT**	Ends a loop
119	**ROT**	rotates stack
120	**RP@**	Shows RTN Pointer
121	**S0**	internal FORTH flag
122	**SFLAG**	Sets User Flag
123	**SIGN**	
124	**SP@**	Shows Stack Pointer
125	**SPACE**	blank space
126	**SWAP**	swapps levels
127	**THEN**	(bool) either
128	**TYPE**	**(**adr, #chars)
129	**U.**	Display unsigned num
130	**UNTIL**	continues Begin/Until
131	**VARIABLE**	Creates named var
132	**VOCABULARY**	Selects which one
133	**WHILE**	Continues if flag true
134	**WORD**	Collects & pack string
135	**XTOF**	*X-reg to pile FORTH*

The following FORTH_79 words are not part of the HP-41's already important set. Some like QUERY and INTERPRET that make FORTH a conversational language will be cruelly missed. They can always be programmed, (in fact note that **DEPTH** and **?DUP** were not present in the original FORTH41 module and now they are included). So here is the list:

***/MOD**
-TRAILING
. ”
BLK
BLOCK
BUFFER
CONVERT
D+
D<
DNEGATE
EMPTY-BUFFER
EXECUTE
EXPECT
FILL
INTERPRET
LITERAL
LOAD
MAX
MIN
MOVE
PICK
QUERY
R@
ROLL
SCR
SAVE-BUFFERS
SPACES
STATE
U*
U/MOD
U<
UPDATE
XOR

Calling HP-41 Functions

In addition to these 135 primitives in ROM, you can add the functions of the HP-41 ROMs (internal functions and modules) directly accessible from the command line – as well as your FOCAL programs in user language accessible with **CALL** 'program name'. Refer to the ‘Structure of a Definition’ section later in the manual.

You can also access functions assigned to any key from the FORTH environment when USER mode is active. They will not be added to the command lien, rather will have an IMMEDIATE execution.

Notably (and probably a good way to get in trouble) you could also call functions from the FORTH Module itself from within the FORTH environment, even those in the “-COMPILER” section using the period instead of the non-keyable goose character.

Getting Started

Let’s assume the following configuration:

HP-41CV 'Halfnut'
Module Clonix with FORTH41 in port 1 (pages **4** & **8**)
Module X-Functions in port 2 (page A)
Module X-Memory in ports 3 and 4

Initialization and smoke testing:

Say there aren’t any user key assignments, then adjust the memory as follows:

SIZE 010		
PRGM	00 REG 309	
PRGM	0.0000	
XEQ ALPHA FORTH ALPHA	OK	*well, hello there!*
BYE R/S	0.0000	

Ok, now we have a positive response, FORTH41 is there ready to go. Just for kicks let’s prepare a trivial demo program for later, simple enough so it’ll only sound a BEEP. Notice the remaining room left for programs after the creation of the FORTH buffer (255 registers long!): 309 – 255 = 54 regs.

PRGM	END. REG 54
<-	00 REG 54
LBL ‘BIP’	01 LBL 'BIP
BEEP	02 BEEP
R/S	03 RTN
GTO ..	00 REG 52
PRGM	0.000

Let’s get back to FORTH and make some sounds calling up the TONE function and the example program from there:

XEQ ALPHA FORTH ALPHA	OK	well, hello there!
CALL BIP R/S	OK	Beep sounds using the FOCAL program
TONE R/S	OK	single tone sounds, always same frequency
BEEP R/S	OK	Beep sounds again using the native function

For starters let’s show off the base conversion words:

1234 HEX . R/S	4D2 OK	*also sets HEX Mode !*
4D2 DECIMAL R/S	OK:	*also sets DEC mode !*
. R/S	1234 OK	our initial value is back

Let’s try our hand with basic arithmetic;

2 3 * R/S	OK:	result is in the top stack level, but not shown
. R/S	6 OK	result dropped in the command line

Contrary to the native HP-41 model, where the LCD is *always* showing the contents of the X-Register, the FORTH command line does not reflect the contents of the top stack level and you need to print it (using ‘ **.** ’) to see it, which also *permanently removes the value from the stack* and sends it to LCD-limbo! (from where there’s no easy way to recover it). This is an important difference between FORTH and the HP-41 that you need to keep in mind if you’re new to FORTH.

Note as well that the "OK:" display helps you by adding a colon if the data stack contains any value, regardless of its depth (i.e. non-empty condition). This is very handy considering that printing gets rid of the top-level register for good, thus is shouldn't be used just to see what's in it.

You can transfer the values in the FORTH top stack level and the X-Register using FTOX and XTOF, but that's of course before you print it to the command line. For example:

```
77 FTOX R/S        OK
BYE R/S            77.0000        back in HP-41 mode
```

When you input characters in the command line they are displayed on the calculator's LCD, which can only hold 12 character, but the characters are also being stored in the ALPHA register of the calculator. What happens when you type a long input string that exceed the 12-char length is that the LCD automatically scrolls to the left the early characters and they come out of view – but they are still in ALPHA, until it is full (24 characters) , signaled by a high-pitch tone. Adding characters after the warning tone will lose the characters at the beginning of the input string and therefore should never be done.

A nice feature of this implementation is the support of the back-arrow key ("<-") to delete the last entered (rightmost) character from the command line. This action *will also scroll the LCD to the right* if more than 12 characters are in the CL, bringing them back to view. You can repeat this action all the way until the LCD (and therefore ALPHA) is empty, or you can use [SHIFT][<-] directly for that.

Bug report: If you reach the 24-char limit in the command line you should not use the back-arrow key to empty in (that is 24 consecutive pressings of it) because that will hang up the calculator. You should use the [SHIFT][<-] instead in this case.

To finish this initial training let's whet our appetite with the ASCII file editor – just a peek, it will be covered later on with more detail.

```
XEQ ALPHA EMDIR ALPHA      DIR EMPTY
<-                         600.0000

ALPHA WORDS41 ALPHA
XEQ ALPHA EDA ALPHA        1, CMD ?_
PRGM                       0.0000
XEQ ALPHA EMDIR ALPHA      WORDS41 A600
```

We see that EDA has used the current value in the X-register to size the ASCII file. More about this later.

Keyboard in FORTH mode :

```
PRGM                       CTRL_
Followed by 1, 2 or 3
```

<Note: to date and after some attempts to reverse-engineer it I have no idea what this CTRL functionality is about. Confusingly enough there's a primitive with similar name but accesses a totally different code… go figure.>

And how about some taste of FORTH?

Here are a few short code snippets to get you warmed-up. This assumes some understanding of the language but not a deep command of it (pun intended). Also bear in mind that each FORTH dialect has its quirks and differences from the standard, and certainly FORTH-41 is not the exception to that rule.

A good reference that got me going is this: https://www.forth.com/starting-forth/

A few trivial examples taken from : *https://wiki.c2.com/?ExampleForthCode*

```
: STAR      ( - -- )     42 EMIT ;
: STARS     ( n -- )     0 DO STAR LOOP CR ;
: SQUARE     ( n -- )    DUP 0 DO DUP STARS LOOP DROP ;
: TRIANGLE ( n -- )      1 DO I STARS LOOP ;
: TOWER      ( n -- )    DUP TRIANGLE SQUARE ;
```

Example: Input a string of (n) characters in the LCD:

This brings up another interesting comparison between FORTH and the HP-41 modes: There's unfortunately no **QUERY** command in the FORTH41 set of primitives, therefore inputting under program control is not as straight-forward as it could have been. The INPT code shown below is a good work-around towards this goal, although it needs to know how many characters are to be entered beforehand.

```
: INPT 0 DO KEY EMIT LOOP ;
```

`6 INPT R/S`   then press six keys to see the characters in the LCD.

Examples: All about Data Stack Depth.

This definition uses the MM> primitive to convert MM format to absolute, needed for the subtraction. It also shows how to read the current stack pointer with SP@ which *must be called before* S0 because such pointer will be updated by calling it (stores the value in the stack):

```
: DEPTH SP@ MM> S0 MM> SWAP - 2/ .  ;
```

Note that after we learn more about the Forth buffer we could also use @ ('fetch') to read the contents of the strategic locations instead of the user variables S0 and SP@ - for sure more cumbersome but it shows the way for several advanced uses when there are no user variables available to do the legwork for us.

```
: DEPTH2 (HEX) 40D1 @ MM> 60D2 @ MM> SWAP - 2/ . ;
```

And here's another way to test if the data stack is empty, this time using a Boolean result:

```
: EMPT (HEX) SP@ S0 = . ;
```

There you have it, although it'll be nicer as an MCODE primitive, right? To be continued…

FORTH Buffer conventions

The FORTH memory is always in the form of a buffer, but a very special one: the address of its first register is somewhere between 0C0 and 0D0, right above the key assignments, but the actionable FORTH memory starts at 0D0 (the first registers if they exist are therefore unused). The first FORTH setting destroys all buffers (but keeps the assignments) and builds the FORTH memory if there is enough space. The following passages in FORTH mode or the execution of the FRAMCHK function re-adjust the size of this buffer by shifting, so that the FORTH constant of the beginning of the memory ('BE') is in register 0D0.

An important feature is to be underscored: all addresses must be handled in MM format (to assist with that there are two decimal conversion primitives for all operations, **HEX** and **OCT**). The 'total' MM format (as opposed to the 'partial' MM format which has been abandoned, you will understand why - *uh??)* is the one you are used to handling in firmware or synthetic programming. *It's represented by 'B-RRR' where B is the byte number (from 0 to 6, each byte uses two digits) and RRR the absolute register number in HEX.* For example, the fourth byte in the Alpha register "M" has 4005 for address under this convention;. The cold-start constant "169" is in bytes 3 & 4 of register 13(c), etc.

Below there's a rudimentary sketch of the Forth buffer structure, refer to the appendices for a more detailed description/view.

```
 MM                                       Adresses ''strategic''
-----------------------------------------
0-0D0          BE                         0-0D0 CTRL1 address to execute
            STATUS                        4-0D0 CTRL2 address to execute
4-0DB                                     8-0D0 CTRL3 address to execute
-----------------------------------------
6-0DB   RETURN STACK                      0-0D1 RP pointer to pile RTN (in Nybble.RRR format)
                 ↓                        4-0D1 SP pointer to pile STACK (in Byte.RRR format)
                 ↑                        8-0D1 NPC return address FEX
8-0ED   DATA STACK                        C-0D1 FIRST 0C0 (constant)
-----------------------------------------
A-0ED   PARM BUFFER                       2-0D2 R0 start of pile R
                ↓
                ↑
C-0FF   INPUT BUFFER
-----------------------------------------
0-100
        PAD
0-109
------------------------------------------
2-109   DICTIONARY
                 ↓
                 ↑
```

A deep understanding of the FORTH buffer is not a mandatory requirement but it will increase your confidence and open the door to more advanced programming. FORTH is a rare language in that it shares some commonalities with lower level languages, but you can only take advantage of those if the buffer structure is well understood.

The Text File Editor

The editor is close to the HP-71B. It is thus easy to create source TEXT files in X-Memory from which new FORTH words will be loaded. When you call if from the FORTH command line, the ASCII file may exist already but if it doesn't it'll be created using the maximum X-mem room available. If you call it from the standard calculator mode, the ASCII file must already exist.

The control characters need to be used first at the '*CMD ?*' prompt. The 9 choices are as follows: T:I:L:D:S:G:B:F:P and are described below. Note that the Text and Insert choices are followed by the text-entering steps, separated by R/S. A Blank R/S returns to the command prompt.

X :File Size, ; ALPHA File Name
ALPHA , XEQ ALPHA EDA ALPHA

List of Editor commands (each should be followed by R/S) :

T	for TEXT write to file. Type your text then R/S for next line. Repeat as needed.
I	for INSERT insertion in front of the current record
L	for LIST listing the file from the current record
D	for DELETE deletion of current recording
S<txt>	<txt> is placed on the record that contains the searched line and this from the current record (syntax: S'TEXT')
G<nn>	to GOTO to record <nn> (G3 is positioned on the third record)
B	for BACKWARD is a step backwards
F	for FORWARD to be a step forward
P	for PRINT either print from the current record
← R/S	return to mode CMD ?
← R/S, ← R/S	return to mode RPN

From the FORTH command line you need to spell the File Name after the EDA primitive. If it doesn't already exist, the editor will create it using all available X-Mem size. If this does not suit your fancy you can create the file first from the FORTH environment before calling the editor, using FTOX, AREDIT (text) , CRFLAS – or let the editor do it for you; here's how:

- Type the File size value, then FTOX to place the value in the X-Register
- Type AREDIT , followed by "FNAME" – the file name you want it to have
- Type EDA FNAME and you're in business.

You can verify that it has been created calling **EMDIR** (also form the FORTH command line) if you want. Be quick using R/S to stop the listing or it'll all dash thru very quickly.

Finally, *loading the ASCII File into the Forth Memory* is just a matter of using the LOADF primitive from the command line, followed by the file name. The definitions included in it will be added to the dictionary block of the FORTH buffer memory after the existing ones.

Note that you could also use the CX Editor **ED**, *but it does not have the capability to input the special characters* used in FORTH… unless of course you're using the extended version **ED+** or **ED$,** available in advanced modules like the WARP_Core.

ASCII File Editor Program Listing

EDA is really a FOCAL program, although it uses special functions like AREDIT and }-EXP that behave differently from what you're used to in that they use input parameters taken from the following program lines, which generally are go-to addresses for MCODE execution. These will be misinterpreted by program listing functions like **PRP** and need to be manually corrected in the printouts.

```
  1  *LBL "EDA"      Editor
  2   CF 21          no stop
  3   SF 25          trap error
  4   }-EXP
  5  <param-1>       create buf
  6  <param-2>       w/ id#14
  7   FC? 25         error'd?
  8   GTO 21         yes, off
  9   ALENG          tst. name
 10   FS? 25         all ok?
 11  GTO 00          yes go on
 12   "NO X-FNS"     nope
 13   GTO 01         show it
 14  *LBL 00
 15   X#0?           Empty?
 16   GTO 00         no, go on
 17   "NAM"          yes abort
 18  *LBL A          show err
 19   >"E ERROR"
 20  *LBL 01
 21   AVIEW
 22   GTO 03         exit
 23  *LBL 00
 24   RDN
 25   FLSIZE         test file
 26   FS? 25         exists?
 27   GTO 00         yes go on
 28   SF 25          trap error
 29   CRFLAS         create it
 30   FC? 25         all ok?
 31   GTO 21         no room,
 32  *LBL 00
 33   POSFL          test type
 34   FS? 25         ASCII ?
 35   GTO 00         yes go on
 36   "TYP"          no, abort
 37   GTO A
 38  *LBL 00
 39   ,
 40   SEEKPTA        get atop
 41   FIX 0
 42   CF 29          no decs
 43   AON            input chrs
 44  *LBL 01
 45   CF 01
 46   XEQ 22
 47   CLA
 48   ARCL X
 49   >", CMD ? "    input cntl
 50   AVIEW          show it
 51   CLA            clear it
 52   CF 19          no insert!
 53   AREDIT         input txt
 54   ALENG          txt length
 55   X=0?           Empty?
 56   GTO 03         yes,
 57   XEQ 06
 58   GTO 01
 59  *LBL 06
 60   64             low end
 61   ATOX
 62   X<=Y?          chr<=64?
 63   TONE 0
 64   X<=Y?
 65   RTN
 66   90             top end
 67   X<=Y?          chr>=90?
 68   TONE 0
 69   X<=Y?
 70   RTN
 71   SF 25
 72   GTO IND Y      dispatch
 73   TONE 0
 74   RTN
 75  *LBL 68         Delete
 76   XEQ 11
 77   DELREC
 78   RTN
 79  *LBL 83         Search
 80   POSFL
 81   INT            record pt
 82   SEEKPT         get there
 83   "NOT FOUND"
 84   X<0?           Found?
 85   AVIEW          no, show
 86   RTN
 87  *LBL 66         Backwrds.
 88   XEQ 22
 89   2
 90   -
 91   X<0?
 92   RTN
 93   SEEKPT
 94   RTN
 95  *LBL 70         Forward
 96   SF 25
 97   GETREC
 98   RTN
 99  *LBL 80         Print
100   XEQ 11
101   SF 25
102   CLA
103   PRA
104   FS? 25
105   G  TO 00
106   "NO PRINTER"
107   AVIEW
108   RTN
109  *LBL 87         Write
110   XEQ 12
111   SF 25
112   SAVEAS
113   FS? 25         did it?
114   RTN            yes, ok
115  *LBL 20         nope
116   "CANNOT"
117   AVIEW
118   RTN
119  *LBL 00
120   SF 21
121   ADV
122   XEQ 12
123   >" :FILE ""
124   -6
125   AROT
126  *LBL 17
127   PRA
128   SF 25
129   GETREC
130   FS? 25
131   GTO 17
132   ADV
133   CF 21
134   RTN
135  *LBL 76         List
136   XEQ 11
137   SF 25
```

```
138  *LBL 08
139  GETREC
140  FC? 25
141  RTN
142  AVIEW
143  GTO 08
144  RTN
145  *LBL 73          Insert
146  SF 01
147  *LBL 84          Type
148  XEQ 11
149  *LBL 09
150  SF 25
151  CLA
152  GETREC
153  RCLPT
154  INT
155  SEEKPT
156  CF 05
157  FC? 25
158  SF 05
159  FC? 01
160  CF 19
161  AVIEW
162  FS? 01
163  CLA
164  AREDIT
165  ALENG
166  X=0?
167  RTN
168  92
169  ATOX             “\”
170  X=Y?
171  GTO 00
172  XTOA
173  -1
174  AROT
175  FS? 05
176  APPREC
177  FS? 05
178  GTO 09
179  SF 25
180  FC? 01
181  DELREC
182  INSREC
183  GTO 09
184  *LBL 00
185  XEQ 06
186  GTO 09
187  *LBL 21
188  "NO ROOM"
189  AVIEW
190  RTN
191  *LBL 03
192  }-EXP
193  <Parm-1>
194  <Param-2>
195  FC? 35
196  AOFF
197  CLST
198  RTN
199  *LBL 71
200  *LBL 11
201  XEQ 22
202  ALENG
203  X=0?
204  ARCL Y
205  ANUM
206  *LBL 02
207  SF 25
208   E
209  -
210  X<0?
211  CLX
212  SEEKPT
213  FC? 25
214  GTO 02
215  RTN
216  *LBL 22
217  RCLPT
218  INT
219  ST- L
220  LASTX
221  X#0?
222  SIGN
223  +
224   E
225  +
226  RTN
227  *LBL 12
228  }-EXP
229  <Parm-1>
230  <Param-2>
231  END
```

Note that this version is slightly different form the original one, I have removed several superfluous LBL and GTO statements as well as consolidated error messages. The changes freed-up enough room to squeeze in another primitive in the ROM-based dictionary.

Another remarkable aspect of EDA is the use of a separate buffer (with id#14) to store the ASCII file name during the processing of the Forth command line, which also utilizes ALPHA for its own purpose. This is done dynamically by the editor, via a clever calling of MCODE snippets directly from the Forth function **}-EXP**. You can refer to the blueprint documents if you’re interested.

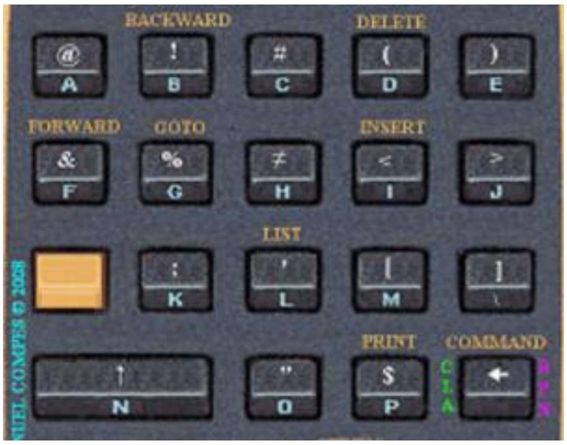
The code below **NXTP** was written by Egan Ford (see *http://sense.net/~egan/forth41/np.forth41*). It calculates the Next Prime from an input number n. Note the subroutine **ISQRT,** used to calculate the Integer Square Root of a number.

The keyboard overlay is added as a reference to guide you in the typing of the characters, since you'll probably use **EDA** to enter the code…

```
DECIMAL
VARIABLE R
VARIABLE M
VARIABLE S
: ISQRT
  M ! 0 R ! 16384 DUP S !
  BEGIN 0> WHILE
    M @ S @ R @ OR < IF
      R @ 2 / R !
    ELSE
      S @ R @ OR NEGATE M +!
      R @ 2 / S @ OR R !
    THEN
    S @ 4 / DUP S !
  REPEAT
  R @
;
: NXTP
  1 + DUP
  2 MOD 0 = IF 1 + THEN
  BEGIN
    DUP ISQRT 1 + 3 DO
    DUP I MOD 0 = IF
      1 LEAVE THEN
    2 +LOOP
    DUP 1 = IF
      DROP 2 + 0
    ELSE DUP THEN
  UNTIL
;
```

```
Get five primes following 1000:
```

```
1000 NXTP NXTP NXTP NXTP NXTP
```



The CRFILE primitive creates a DATA file named "@FILE" of type "4" with all the available room in X-Mem. Not sure what exactly this is used for yet, so further investigation is needed. Note that the file type is incompatible with the Matrix files created by the CCD or Advantage/SandMatrix modules/

The LOADIF primitive is very useful to load FORTH code. Below is an example of utilization.

The FOCAL routine below creates an ASCII file names NXTP with FORTH Code used in previous example. Once created you only need to use "LOADIF NXTP" to have the two new definitions added to the RAM dictionary, it doesn't get any easier.-

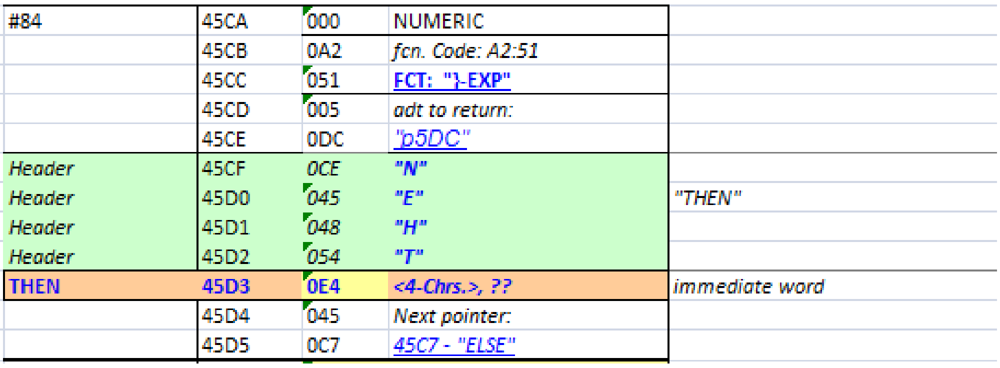
```
*LBL "NXTP"
"NXTP"
SF 25
PURFL
CF 25
50
CRFLAS
,
SEEKPTA
"DECIMAL"
APPREC
"VARIABLE R"
APPREC
"VARIABLE M"
APPREC
"VARIABLE S"
APPREC
": ISQRT"
APPREC
"M ! 0 R ! 16384"
>" DUP S !"
APPREC
"BEGIN 0> WHILE"
APPREC
"M @ S @ R @ OR "
>"< IF"
APPREC
"R @ 2 / R !"
APPREC
"ELSE"
APPREC
"S @ R @ OR NEGA"
>"TE M +!"
APPREC
"R @ 2 / S @ OR "
>"R !"
APPREC
"THEN"
APPREC
"S @ 4 / DUP S !"
APPREC
"REPEAT"
APPREC
"R @ ;"
APPREC
": NXTP 1 + DUP"
APPREC
"2 MOD 0 = IF 1 "
>"+ THEN"
APPREC
"BEGIN DUP ISQRT"
APPREC
" 1 + 3 DO DUP I"
APPREC
" MOD 0 = IF 1 "
APPREC
"LEAVE THEN 2 +L"
>"OOP"
APPREC
"DUP 1 = IF DROP"
>" 2 + 0"
APPREC
"ELSE DUP THEN U"
>"NTIL ;"
APPREC
END
```

Structure of a definition.

FORTH words are coded in a similar way to user programs and therefore read backwards (decreasing addresses). The definitions are stored upside down so that the memory size can be changed (with GROW), therefore the end address of the program has to be stored in the definition.

The structure is therefore as follows:

- LFA (Link): address of the NFA of the previous definition,
- NFA (Name): A0+word length+40 if the word is immediate,
- name in ASCII, with the most significant bit of the last character at 1,
- CFA (Code): 4 bytes to be executed (upside down) to launch the definition,
- PFA (Param): zero if the definition does not ask for an alpha parameter,
- program coded backwards.



#84	45CA	000	NUMERIC	
	45CB	0A2	*fcn. Code: A2:51*	
	45CC	051	FCT: "}-EXP"	
	45CD	005	*adt to return:*	
	45CE	0DC	*"p5DC"*	
Header	45CF	*0CE*	*"N"*	
Header	45D0	*045*	*"E"*	*"THEN"*
Header	45D1	*048*	*"H"*	
Header	45D2	*054*	*"T"*	
THEN	45D3	0E4	*<4-Chrs.>, ??*	*immediate word*
	45D4	045	*Next pointer:*	
	45D5	0C7	*45C7 - "ELSE"*	

As the interpreter does not know the length of the definition when it starts writing, it starts at the end of the FORTH buffer and continues upwards, then it takes care of the block transfer and finishes the definition. This is why the D, and C D, primitives have been defined. They are analogous to , and C, although here they decrement instead (so the "d" does not make a reference to double words at all!).

In order to speed up the editor, ***the compilation of the words is done at the pressing of SPACE***. These words are *compiled in the input buffer* and when a word requires an alpha parameter, this string is carefully *set aside in the parameter buffer*. Apart from that, the entered characters are added to the alpha register (*which is therefore theoretically limited to 24 characters*). The correction key obviously decompiles the entered words.

Sources :

Micro-Revue n°5 from March/April 1985
Letter from Jean-Pierre Baudoin (T131) from 12/11/85 addressed to Micro-Revue readers.
FORTH Handy Reference from FORTH Interest Group
Programmer le FORTH by Robert Van Loo aux Editions Marabout

Appendix.

Table of primitives *arranged by search sequence* and length-threads:

5	4	3	2	1
SPACE	EMIT	BYE	<#	#
OCTAL	TYPE	DUP	#S	├
ABORT	HOLD	ROT	#>	+
DOES>	SIGN	MM>	U├	@
BEGIN	DROP	>MM	C@	!
UNTIL	SWAP	MOD	C!	?
+LOOP	OVER	ABS	+!	-
LEAVE	/MOD	AND	>R	*
ITYPE	FIND	NOT	R>	/
ALLOT	CTRL	PAD	*/	=
WHILE	HERE	+IN	0=	<
COUNT	BASE	SP@	0<	>
CMOVE	EXIT	RP@	0>	"
LOADF	QUIT	HEX	1+	,
SFLAG	PARM	CD,	1-	:
CFLAG	LOOP	KEY	2+	]
ALPHA	THEN	EDA	2-	[
CRBUF	ELSE	M1+	OR	,
DEPTH	WORD	M1-	C,	0
	CALL	M2+	S0	I
	FTOX	M2-	R0	J
	XTOF		D,	(
	GROW		DO	)
	FLAG		IF	
			CR	
	BUF0		2*	
	?DUP		2/	

7	6	8
0BRANCH	NEGATE	VARIABLE
COMPILE	BRANCH	CONSTANT
CONTEXT	CREATE	
CURRENT	FORGET	
DECIMAL	.STATE	
DESTROY	REPEAT	
	AREDIT	
	CRFILE	

9	10	11
IMMEDIATE	VOCABULARY	DEFINITIONS

NULL	
#	*thread-1*
<#	*thread-2*
BYE	*thread-3*
EMIT	*tHread-4*
SPACE	*thread-5*
NEGATE	*thread-6*
0BRANCH	*thread-7*
VARIABLE	*thread-8*
IMMEDIATE	*thread-9*
VOCABULARY	*thread-10*
DEFINITIONS	*thread-11*

User Variables.

There are several user variables predefined in FORTH41. The table below shows their names, buffer location and nominal values for the examples used. Note that some of them can be accessed with Forth primitives as well.

Name	Primitive	Location	Value	Comment
PNT: S	**SP@**	4.0D1	8.0DB	Current Stack Pointer
PNT: R	**RP@**	0.0D1	8.0ED	Current RTN pointer
PNCX	n/a	6.0D1	RPN PC	PNC RTN adr FEX
CURRENT	**CURRENT**	8.0D5	8.10B	Vocabulary used to store word
CONTEX	**CONTEX**	4.0D5	810B	Vocabulary to search first
HERE	**HERE**	5.0D6	xxxx	Next available Dictionary location
BASE	**BASE**	0.0D5	10 / 0F	Selected Base
PAD	**PAD**		100	Start of PAD area
TOP:S	**S0**		8.0ED	ADR of Parameter Stack start
TOP:R	**R0**		8.0DB	SADR of RTN stack start
TOP:D	n/a		2.109	Start of Dictionary
IN>	**>IN**	2.0D3	????	Current INPUT pointer

The buffer id# is 0xD (13 dec). The buffer Size must be at least 70 registers and can be up to 256 registers large 0 in Hex that is [46, FF]. Its actual size will be determined by the available space in the I/O area at the time of creation by executing FORTH the first time. You can also create the buffer manually with CRFRAM, which requires the buffer size in the X register as input (must be between [70, 256] as mentioned above).

The buffer header is located just above the key assignments, therefore somewhere between 0x0C0 with no KA and 0x0CF with the USER keyboard fully loaded. So in theory the existing key assignments are respected, but once the buffer is created no additional ones can be made. Notably, even when not occupied by KA, buffer registers up to 0CF included are not used at all; thus 0D0 is the first one of the Control Registers block – ranging from 0D0 to 0DB . The first control register is marked with the byte «**BE**» in D0<13 :12>

Buffer Control block

By far the most strategic, this block holds pointers to key addresses reflecting the current status of the forth buffer. In general *Stack Pointers use B.RRR format*, 7 bytes per register, whereas *RTN Pointers use N.RRR format*, 14 nybbles per register

Here's the Control Block detail – which spans from 0.0D0 tp 8.0DB: Note the locations of the user variables, as well as the default values for other fields still unidentified.

Register		6		5		4		3		2		1		0			
DEC	HEX	D	C	B	A	9	8	7	6	5	4	3	2	1	0		
208	**0D0**	***"B"***	***"E"***	**0000 - CTL:3**				**0000 - CTL:2**				**0000 - CTL:1**				T	
209	**0D1**	C	2	**0200 - NPC:X**				**80ED - PNT:S**				**C0DB - PNT:R**				Z	
210	**0D2**	**2109 - TOP:D**				**80ED - TOP:S**				**80DB - TOP:R**				0	0	Y	
211	**0D3**	D	0	**A0ED - INPT START**				**A0FF - INPT END**				A0FF - INPUT				X	
212	**0D4**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	L	
213	**0D5**	0	0	**810B - CURRENT**				**810B - CONTEXT**				**0010 - BASE**				M	***Control Regs***
214	**0D6**	01BF -				01BF -				**610C - HERE**				**0100 --**		N	
215	**0D7**	**- PAD**		0	0	0	0	0	0	0	0	0	0	0	0	O	
216	**0D8**	40ED -				614E -				41B4 -				0	0	P	
217	**0D9**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Q	
218	**0DA**	0/F RTN:3					0/F RTN:4									|-	***RTN Stack***
219	0DB	0	0	0	0	0/F RTN:1					0/F RTN:2					a	

In all likelihood all these fields will hold dynamic pointers to buffer locations, so the FORTH routines know where to take commands and parameters from, as well as where to put the results in after the actions are performed. Some general knowledge of the workings of FORTH interpreter and compiler theory are going needed to guide our investigations, but even with that under our belt determining the role and purpose of each one of them is going to require some sleuth work and plenty of trial and error.

Buffer Parameter Block.

The Parameter block occupies an area of 18 registers, from 0x0DB to 0x0ED both partially included. As each parameter takes 2 bytes, that means we can fit 7 parameters every two-registers, with a theoretical maximum of 7 x (18/2) = 63 parameters.

As parameter values are being typed in the command line (separated by spaces) two things occur: (1) they are entered in the data block area of the buffer, starting at 8.0ED, with 16-bits per value (2 bytes), and (2) The current stack pointer value in **PNT:S** (at 4.0D1) is decreased by two bytes, marking the next available location.

This means that :

- the top of the Forth stack has the *first value entered*, always located at 8.0ED, (adr stored in **S0**)

- the bottom of the Forth stack has the *most recent (last) value entered*, which is located at the address pointed by **PNT:S** and can be accessed by the word **SP@**

- An 'STACK_EMPTY' condition exists when **PNT:S** equals **S0**, and in fact this is how FORTH41 determines it.

- When stack values are printed into the LCD the current stack pointer **PNT:S** is decreased, but for a faster implementation its value is not cleared from the buffer - It'll simply be overwritten when the depth of the stack reaches that level again.

The picture below shows the complete Parameter Block after entering 16 hex values in the command line, from FFFF to 0 – decreasing one at a time:

Register		6		5		4		3		2		1		0		
DEC	HEX	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
219	0DB	.....	.....	.....	.....											a
220	0DC	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	b
221	0DD	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	c
222	0DE	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	d
223	0DF	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	e
224	0E0	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	|-
225	0E1	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
226	0E2	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
227	0E3	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
228	0E4	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
229	0E5	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
230	0E6	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
231	0E7	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
232	0E8	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	
233	0E9	2	2	2	2	1	1	1	1	0	0	0	0	.....	.....	
234	0EA	6	6	5	5	5	5	4	4	4	4	3	3	3	3	
235	0EB	9	9	9	9	8	8	8	8	7	7	7	7	6	6	
236	0EC	D	D	C	C	C	C	B	B	B	B	A	A	A	A	
237	0ED					F	F	F	F	E	E	E	E	D	D	

Parameter Stack

For this example we have :

```
S0 U.     =>   8.0ED   OK
SP@ U.    =>   0.0E9   OK
```

Buffer Input Block.

The Input block holds the information entered in the command line, and it's being stored there with each pressing of the SPACE key to separate two commands or parameters. It also uses 18 registers, which allows for up to 18 x 7 = 126 characters ; or in other words 5-times a full ALPHA register worth.

Control register 0D3 holds three pointers that define the block boundaries, as follows:

- INPUT_START at 0xA0ED saved in D3<B :8>
- INPUT_END at 0xA0FF saved in D3<7 :4>), as well as
- Pointer to the current data entry address, saved in D3<3 :0>.

This last value can always be recalled using the primitive IN\ as well.

What goes into the Input block ? Basically, a direct copy of the command line, including primitive names, parameter values, variable names, etc. It's hard to spy on this block's contents because it gets deleted upon the CALC_ON event, so the only way to pry on it is on the fly, by interrupting the MCODE routines that read/write from/to it as they're being run.

237	0ED	4	9	0	0										
238	0EE	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
239	0EF	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
240	0F0	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
241	0F1	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
242	0F2	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
243	0F3	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
244	0F4	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
245	0F5	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
246	0F6	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
247	0F7	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
248	0F8	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
249	0F9	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
250	0FA	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....
251	0FB	- U.		0	0	0	0	A2:4F - }-EXB				0	0	0	0
252	0FC	A2:65 - EMIT				0	0	0	0	A2:4C - }-COMP				A2:64 -	
253	0FD	- }-EXB		A2:51 - }-EXP				05C9 - "key" adr				A2:4C - }-COMP			
254	0FE	A2:4F - }-EXB				0	0	0	0	0	0	0	0	A2:4F -	
255	0FF	0	0	A2:4C - }-COMP				A2:51 - }-EXP				019B - "BYE" adr			

Input Buffer

For command Line

Note: you may wonder about resorting to some hacking tools like the **VRG** function in the AMC_OS/X module (not the library#4-based version for obvious reasons) to see the contents of the registers, since this can be called from the Forth environment using the USER direct access… and it's a good idea that will be exploited later on.

Buffer Dictionary Block.

This section of the Forth buffer is where the definitions, variables and constants are stored. It starts right below the PAD area, in register 0x109 (in nybble 2109 to be precise). In fact, the first three registers are populated by the Forth buffer creation routines, entering 26 bytes as shown below:

Register		6		5		4		3		2		1		0	
DEC	HEX	D	C	B	A	9	8	7	6	5	4	3	2	1	0
265	109	*52 -"R"*		*4F = "O"*		*46 = "F"*		*E5 - <5-Chrs.>*		0	0	0	0	PAD	PAD
266	10A	00 - NUM		A2:4E - }-EX				810B - Next ADDR				*C8 = "H"*		*54 = "T"*	
267	10B	- ????		A114 - LAST NAME				A2:51 - }-EXP				0139 - LAST REG			
268	10C	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	A081 -	

This constitutes the header of this block, so there's a FORTH word in there as well (so that you can actually type FORTH in the command line, although I'm not sure why would that be needed?). Besides the name, the block header has pointers to the end of the block (0x0139 in here), and to the end of the occupied area (0xA114 in this example) – which is the next available address for the next definition. It ends with two bytes with unclear purpose (0xA081), not a viable address, nor a function code…

We can write this sequence in the form of a proper definition, as follows: (remember the definition is written backwards, so the addresses are ascending from the bottom of it):

B.NNN	Code		
010C	**0A0**	**????**	*Unknown purpose*
610B	**081**	**????**	*Not an addr or fnc. code*
510B	0A1	*Last Name addr*	*End of occupied block*
410B	014	*Available for next*	*Address: 0xA114*
310B	*0A2*	*XROM 09,17*	
210B	*051*	*A2:51*	*}-EXP*
110B	001	*Last used Reg#*	*Where the block ends:*
010B	039	*Within the block*	*Address: 0x139*
610A	000	NUMERIC	
510A	*1A2*	*XROM 09,14*	
410A	*04E*	*A2:4E*	*}-EX*
310A	081	*NAME addr*	In N.RRR format
210A	00B	Next definition	0x810B
110A	*0C8*	***"H"***	
010A	*054*	***"T"***	*"FORTH"*
6109	*052*	***"R"***	
5109	*04F*	***"O"***	
4109	*046*	***"F"***	
3109	*0E5*	*#Chars*	*Immediate, <5-Chrs>*
2109	000	*Previous Name*	
1109	000	*address*	*Beginning of the block*
0109	*000*	*PAD*	
6108	*000*	*PAD*	

For the next steps, let's go easy just declaring four variables which will be named "ANGEL", LUJAN", "TEST", and "XX" entered in that order. The sequence will be loaded following the section header, see below the snippets created for each variable:

Register		6		5		4		3		2		1		0	
DEC	HEX	D	C	B	A	9	8	7	6	5	4	3	2	1	0
268	10C	41 = "A"		A5 - <5.Chrs.>		6109 - Prev. NAME				0	0	0	0	VARIABLE "ANGEL"	
269	10D	}-EX		C10E - Next ADDR				CC = "L"		45 = "E"		47 = "G"		4E = "N"	
270	10E	.....	.....	A2:53 - }-RNT				20DA = RTN1				00 - NUM		A2:4E	
270	10E	0	0	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	VARIABLE "LUJAN"	
271	10F	4A - "J"		55 - "U"		4C = "L"		A5 - <5.Chrs.>		A10C - Prev. NAME				0	0
272	110	00 - NUM		A2:4E - }-EX				8111 - Next ADDR				CE = "N"		41 = "A"	
273	111	.....	.....	.....	.....	.....	.....	A2:53 - }-RNT				20DA = RTN2			
273	111	- Prev. NAME		0	0	0	0	.....	.....	.....	.....	.....	.....	VARIABLE "TEST"	
274	112	- Next ADDR		D4 = "T"		53 = "S"		45 = "E"		54 = "T"		A4 - <4-Chrs.>		610F -	
275	113	- }-RNT		20DA = RTN3				00 - NUM		A2:4E - }-EX				2114 -	
276	114	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	.....	A2:53 -	
276	114	58 = "X"		A2 - <2-Chrs.>		2112 - Prev. NAME				0	0	0	0	VARIABLE "XX"	
277	115	- RTN4		00 - NUM		A2:4E - }-EX				6116 - Next ADDR				D8 = "X"	
278	116	.....	.....	.....	.....	.....	.....	.....	.....	A2:53 - }-RNT				20DA -	

In a similar move as done before we can write the definition for the first variable in "vertical" way, which works better to identify the fields within it. The structure is the same for all variables, practically identical to the ROM-based primitives included in the module. The primitives involved are **}-EX** and **}-RNT** at the end, which appears to signal the RTN address for this variable. Bottom line is each definition takes 12 bytes plus the name length, so 17 bytes in this case:

510E	*1A2*	*XROM 09,19*	
410E	*053*	*A2:53*	*}-RNT*
310E	120	*RTN addr*	
210E	0DA	*This definition*	
110E	000	NUM / ALPHA	
010E	*1A2*	*XROM 09,14*	
610D	*04E*	*A2:4E*	*}-EX*
510D	1C1	*NAME adr*	
410D	00E	*Next definition*	
310D	*0CC*	**"L"**	
210D	*045*	**"E"**	
110D	*047*	**"G"**	
010D	*04E*	**"N"**	
610C	*041*	**"A"**	
510C	*A05*	*#Chars*	*<5-Chrs.>*
410C	1A1	*Previous Name*	
310C	00C	*address*	
210C	*000*	*blank*	
110C	*000*	*blank*	

Remarks:

Using the variable name returns the *definition start* (***end*** *address) in the buffer*, thus:

```
ANGEL U. R/S    =>   C10E   OK
LUJAN U. R/S    =>   8111   OK
TEST U. R/S     =>   2114   OK
XX U. R/S       =>   6116   OK
```

The field at 0x0DA points at the (beginning of the name of the) last definition in the dictionary, as it can be seen by checking the contents after declaring each one of the variables:

ANGEL: now pointing at A.10C

0DA	0	00:00	0	00:00	00:00		
0DB	0	0	0	0	0	00:00	0	**A10C – beginning of DEF-1 name**

LUJAN: now pointing at 6.10F

0DA	0	00:00	0	00:00	A2:4F - **}-EXB**			
0DB	0	0	F	F	0	00:00	0	**610F – beginning of DEF-2 name**

TEST: now pointing at 2.112

0DA	0	00:00	0	0:00	A2:4F - **}-EXB**			
0DB	0	0	F	F	0	0:00	0	**2112 – beginning of DEF-3 name**

XX: now pointing at A.114

0DA	0	00:00	0	0:00	A2:4F - **}-EXB**			
0DB	0	0	F	F	0	0:00	0	**A114 – beginning of DEF-4 name**

For the next step let's be brave and use a simple definition to see how it gets saved in the buffer. Say we create the word INP to put in the LCD the character entered in a KEY action:

```
: INP KEY EMIT ;
```

Here's the buffer contents after this action, located right after the last variable ("XX"):

278	116	A114 - Prev. NAME	0	0	0	0	.....	.....	.....	.....	**: INP KEY EMIT ;**
279	117	- }-EX	**8119 - Next ADDR**	*D0 = "P"*	*4E = "N"*	*49 = "I"*	*A3 - <3-Chrs.>*				
280	118	A2:65 - EMIT	A2:4F - }-EXB	0	0	0	0	A2:4E -			
281	119	.....	.....	.....	.....	.....	.....	A2:51 - }-EXP	05C9 - "KEY" ADDR		

And the corresponding update made to the 0x0DA pointer, now pointing at 0.117

0DA	0	00:00	0	0:00	A2:4F - }-EXB			
0DB	0	0	F	F	0	0:00	0	**0117 – beginning of DEF-5 name**

Execution of a word saved in the dictionary

https://www.forth.com/starting-forth/1-forth-stacks-dictionary/

When you define a new word, Forth translates your definition into dictionary form and writes the entry in the dictionary. This process is called "compiling."

For example, when you enter the line

```
: STAR 42 EMIT ; R/S   => OK
```

the compiler compiles the new definition into the dictionary. The compiler does *not* print the asterisk.

Once a word is in the dictionary, how is it executed? Let's say you enter the following line directly at your terminal (not inside a definition):

STAR 30 SPACES↵

This will activate a word called INTERPRET, also known as the "text interpreter." The text interpreter scans the input stream, looking for strings of characters separated by spaces. When a string is found, it is looked up in the dictionary.

If the word is in the dictionary, it is pointed out to a word called EXECUTE, which executes the definition (in this case an asterisk is printed). Finally, the interpreter says everything's "OK"

If the interpreter cannot find the string in the dictionary, it calls the number-runner (called NUMBER). NUMBER knows a number when he sees one. If NUMBER finds a number, he runs it off to a temporary storage location for numbers.